

高等学校教材·计算机科学与技术

人工智能（AI）程序设计

（面向对象语言）

雷英杰 邢清华 王涛 等 编著

清华大学出版社
北 京

内 容 简 介

本书主要介绍人工智能的基础知识和应用于人工智能与专家系统领域的面向对象逻辑程序设计语言 Visual Prolog 等内容。

第 1 部分主要介绍人工智能的基础知识、知识的表示方法以及 AI 的编程基础。第 2 部分介绍 Visual Prolog 的编程基础, 主要包括 Visual Prolog 的类与对象机制、程序结构、GUI 编程、逻辑层编辑、数据层编程、CGI 编程等。第 3 部分介绍 Visual Prolog 的语言特性, 主要包括 Visual Prolog 语言元素、Visual Prolog 数据元素、Visual Prolog 程序元素以及 Visual Prolog 与其他编程语言接口等。

本书适合于计算机课程体系中智能类课程的教学, 也可供有关专业的师生和科技人员参考。

版权所有, 翻印必究。举报电话: 010-62782989 13501256678 13801310933

本书封面贴有清华大学出版社防伪标签, 无标签者不得销售。

本书防伪标签采用清华大学核研院专有核径迹膜防伪技术, 用户可通过在图案表面涂抹清水, 图案消失, 水干后图案复现; 或将表面膜揭下, 放在白纸上用彩笔涂抹, 图案在白纸上再现的方法识别真伪。

图书在版编目(CIP)数据

人工智能(AI)程序设计(面向对象语言)/雷英杰, 邢清华, 王涛等编著. —北京: 清华大学出版社, 2005.3

(高等学校教材·计算机科学与技术)

ISBN 7-302-10429-8

I. 人… II. ①雷… ②邢… ③王… III. ①人工智能-高等学校-教材 ②PROLOG 语言-程序设计-高等学校-教材 IV. ①TP18 ②TP312

中国版本图书馆 CIP 数据核字(2005)第 008320 号

出 版 者: 清华大学出版社

<http://www.tup.com.cn>

社总机: 010-62770175

地 址: 北京清华大学学研大厦

邮 编: 100084

客户服务: 010-62776969

组稿编辑: 丁 岭

文稿编辑: 霍志国

印 装 者: 世界知识印刷厂

装 订 者: 三河市金元装订厂

发 行 者: 新华书店总店北京发行所

开 本: 185×260 印张: 27.75 字数: 677 千字

版 次: 2005 年 3 月第 1 版 2005 年 3 月第 1 次印刷

书 号: ISBN 7-302-10429-8/TP·7081

印 数: 1~3000

定 价: 38.00 元

前 言

Prolog 语言是人工智能与专家系统领域最著名的逻辑程序设计语言。Visual Prolog 指可视化逻辑程序设计语言，是基于 Prolog 语言的可视化集成开发环境，是 Prolog 开发中心（PDC）最新推出的基于 Windows 环境的智能化编程工具，其语言特性符合相应的国际标准 ISO/IEC 13211-1:1995。

Visual Prolog 是当今新一代开发智能化应用的强有力工具，它还支持基于网络的开发、数据库、多媒体、与 C 语言集成等。Visual Prolog 在美国、加拿大、西欧、澳大利亚、新西兰、日本、韩国、新加坡等发达国家和地区十分流行，是国际上研究和开发智能化应用的主流工具之一。目前，中国在智能化领域的教学、研究、开发及应用正在迎来一个蓬勃发展的新时期，拥有较多的群体，对这种工具软件的需求已经逐渐显现出来。国内已有不少 Visual Prolog 用户，一个 Visual Prolog 群体正在逐渐形成。预计不久的将来，在国际上已经十分流行的最新版本的可视化逻辑程序设计语言 Visual Prolog 将会在国内广泛流行，并将迅速成为中国研究和开发智能化应用的主流工具。

Visual Prolog 具有模式匹配、递归、回溯、对象机制、事实数据库和谓词库等强大功能。它包含构建大型应用程序所需要的一切特性：图形开发环境、编译器、连接器和调试器、支持模块化和面向对象程序设计、支持系统级编程、文件操作、字符串处理、位级运算、算术与逻辑运算，以及与其他编程语言的接口。

Visual Prolog 包含一个大型库，捆绑了大量的 API 函数，包括 Windows GUI 函数族、ODBC/OCI 数据库函数族和因特网函数族（socket, FTP, HTTP, CGI 等）。这个开发环境全部使用 Visual Prolog 语言写成，而且包含对话框、菜单、工具栏等若干编码专家和图形编辑器。Visual Prolog 支持 Windows 9x/Me/NT/2000/XP, OS/2, Linux 和 SCOUNIX 等操作系统。

Visual Prolog 非常适合于专家系统、规划和其他 AI 相关问题的求解，是智能程序设计语言中具有代表性且应用较多的一种语言。由于这种语言很适合表达人的思维和推理规则，在自然语言理解、机器定理证明、专家系统等方面得到了广泛的应用，在智能程序设计语言中占有相当重要的地位。Visual Prolog 不仅是优秀的智能化应用开发工具，而且与 SQL 数据库系统、Visual C++或其他 C++开发系统、Visual Basic, Delphi 或 Visual Age 等编程语言一样，已经成为适用于任何应用领域的强有力的通用开发工具。

智能化是当前计算机、自动化、通信、管理等信息科学技术领域中的新方法、新技术、新产品的重要发展方向与开发策略之一。信息处理的智能化与信息社会对智能的巨大需求是人工智能发展的强大动力。人工智能与专家系统曾取得过许多令人瞩目的成果，也走过不少弯路，经历过不少挫折。近几年来，随着计算机及网络技术的迅猛发展，特别是因特网的大规模普及，人工智能与专家系统的研究再度活跃起来，并正向更为广阔的领域发展。围绕人工智能与专家系统的研究和应用开发也迎来一个蓬勃发展的新时期。因此，引进与消化国际上已经广泛流行的功能强大和通用的智能程序设计语言、工具与环境，对于中国

开发智能化应用系统十分必要。鉴于国内已有许多用户在使用 Visual Prolog，而这方面的中文资料比较缺乏，我们编写了本书，系统介绍了基于 Visual Prolog 的 AI 程序设计的功能特点、编程方法与技术，相信对于开发智能化软件有启迪作用，也希望对国内在这一领域的教学、研究及智能化应用水平的提高起到良好的促进作用，且有益于国内同行在这一领域与国际主流保持一致。

王宝树教授在百忙之中审阅了全书，并提出大量有益的意见和建议。作者非常感谢西安电子科技大学计算机学院的良好氛围和条件支持，特别要感谢王宝树教授、周利华教授、李荣才教授等的指导和鼓励，还要感谢空军工程大学计算机工程系吕辉教授、李续武教授等的支持和帮助，真诚感谢清华大学出版社的大力支持和丁玲老师及霍志国老师辛勤的工作，正是由于这众多的帮助和支持才使本书得以呈献给读者。在本书编写过程中，还参阅了有关资料，在此对这些资料的作者们深表感谢。

逻辑程序设计与面向对象两大主流技术的融合是 Visual Prolog 的一大特点。本书在编写过程中，充分考虑了 CC2001/CCC2002 课程体系的要求，从而适合于计算课程体系中智能类课程的教学，也可供有关专业的师生和科技人员参考。

参加本书资料性工作的还有博士生王晶晶、陈东峰，硕士生李兆渊、路艳丽、项军、汪竞宇、吉波、刘佳昀、孙晨、庄瑾等，在此一并表示感谢。

需要特别指出，虽然作者竭尽所能，精心策划章节结构和内容编排，详细测试书中的每个实例，尽可能简明而准确地表述其意，但限于水平和资料，书中的错误和不足之处在所难免，恳请读者不吝指正。

作 者
2005 年

目 录

第 1 部分 基础知识

第 1 章 人工智能概述	2
1.1 人工智能的概念	2
1.1.1 人工智能	2
1.1.2 为什么要研究人工智能	3
1.1.3 人类智能的计算机模拟	4
1.2 人工智能的研究目标	7
1.3 人工智能研究的基本内容及特点	9
1.3.1 人工智能研究的基本内容	9
1.3.2 人工智能的研究途径与方法	10
1.3.3 人工智能研究的主要特点	12
1.4 人工智能的研究领域	14
1.4.1 经典的人工智能研究领域	14
1.4.2 基于脑功能模拟的领域划分	24
1.4.3 基于实现技术的领域划分	28
1.4.4 基于应用领域的领域划分	28
1.4.5 基于应用系统的领域划分	33
1.4.6 基于计算机系统结构的领域划分	34
1.4.7 基于实现工具与环境的领域划分	35
1.5 人工智能的基本技术	35
1.5.1 推理技术	35
1.5.2 搜索技术	36
1.5.3 知识表示与知识库技术	37
1.5.4 归纳技术	37
1.5.5 联想技术	37
1.6 人工智能的产生与发展	38
1.6.1 人工智能学科的产生	38
1.6.2 符号主义学派	38
1.6.3 连接主义学派	40
1.6.4 人工智能的发展趋势	41
1.6.5 中国人工智能的研究与发展	42
本章小结	42

习题 1	44
第 2 章 知识表示方法	45
2.1 知识的基本概念	45
2.1.1 知识层次	45
2.1.2 知识的属性	46
2.1.3 知识分类	47
2.1.4 知识表示	48
2.2 一阶谓词逻辑表示法	51
2.2.1 命题与真值	51
2.2.2 论域和谓词	51
2.2.3 谓词公式与量词	52
2.2.4 谓词逻辑表示方法	53
2.2.5 谓词逻辑表示方法的 BNF 描述	54
2.2.6 谓词逻辑表示方法的特点	54
2.3 产生式表示法	55
2.3.1 产生式	55
2.3.2 产生式系统	56
2.3.3 产生式表示法的特点	61
2.3.4 产生式表示法与其他知识表示方法的比较	61
2.4 语义网络表示法	62
2.4.1 语义网络的基本结构	62
2.4.2 语义网络的知识表示	62
2.4.3 语义网络与 Prolog	64
2.4.4 语义网络的求解流程	65
2.4.5 基本的语义关系	65
2.4.6 语义网络表示法的特点	67
2.4.7 语义网络法与其他知识表示方法的比较	67
2.5 框架表示法	68
2.5.1 框架的基本结构	68
2.5.2 框架的 BNF 描述	70
2.5.3 框架系统中的预定义槽名	71
2.5.4 框架系统的问题求解过程	72
2.5.5 框架系统的程序语言实现	73
2.5.6 框架系统的特点	73
2.6 脚本表示法	73
2.6.1 概念依赖理论	74
2.6.2 脚本的结构	74
2.6.3 脚本的推理	75

2.6.4 脚本表示法的特点	76
2.7 过程表示法	76
2.7.1 表示知识的方法	77
2.7.2 过程表示的问题求解过程	78
2.7.3 过程表示的特点	79
2.7.4 过程性与说明性表示方法的比较	79
2.8 Petri 网表示法	79
2.8.1 Petri 网的基本概念	80
2.8.2 表示知识的方法	80
2.8.3 Petri 网表示法的特点	81
2.9 面向对象表示法	81
2.9.1 面向对象的基本概念	81
2.9.2 面向对象的基本特征	83
2.9.3 面向对象的知识表示	83
2.9.4 面向对象表示方法的特点	84
2.10 状态空间表示法	85
2.11 问题归约表示法	85
本章小结	86
习题 2	86
第 3 章 AI 编程基础	88
3.1 命题逻辑	88
3.1.1 命题	88
3.1.2 命题定律	90
3.1.3 范式	92
3.1.4 命题逻辑的推论规则	94
3.1.5 命题逻辑的局限性	94
3.2 一阶谓词逻辑	95
3.2.1 谓词	95
3.2.2 量词	96
3.2.3 谓词逻辑的合式公式	97
3.2.4 自由变元与约束变元	97
3.2.5 谓词公式的解释	98
3.2.6 含有量词的等价式和蕴含式	99
3.2.7 谓词逻辑中的推论规则	101
3.2.8 谓词公式的范式与斯柯林标准形	102
3.3 产生式系统	104
3.3.1 产生式系统的基本组成	104
3.3.2 产生式系统的基本过程	106

3.3.3	基于产生式系统的具体问题建模	107
3.3.4	产生式系统的类型	108
3.3.5	产生式系统的搜索策略	109
3.3.6	两种典型的产生式系统	112
3.4	专家系统	117
3.4.1	专家系统的概念与组成	117
3.4.2	专家系统的类型	120
3.4.3	专家系统的特点	122
3.4.4	专家系统的开发工具	123
3.4.5	新一代专家系统研究	124
	本章小结	126
	习题 3	127

第 2 部分 编程指南

第 4 章	Visual Prolog 概述	130
4.1	Visual Prolog 6 概述	130
4.2	Visual Prolog 6 基本特性	131
4.2.1	语言特性	131
4.2.2	图形化开发环境	132
4.2.3	编译器	132
4.2.4	链接器	132
4.2.5	调试器	132
4.3	创建项目	133
4.4	建立项目	134
4.5	浏览项目	135
4.6	开发项目	137
4.7	调试项目	140
	本章小结	142
	习题 4	142

第 5 章	Prolog 基础	143
5.1	Horn 子句逻辑	143
5.2	Prolog 推理机	145
5.3	扩展家庭定理	146
5.4	Prolog 是一种编程语言	147
5.5	程序控制	148
5.5.1	失败	149
5.5.2	回溯	149

5.5.3	改进家庭定理	151
5.5.4	递归	152
5.5.5	副效应	153
5.5.6	小结	154
5.6	Prolog 算符	154
5.6.1	算符	155
5.6.2	深入理解算符	156
5.6.3	算符与谓词	158
5.6.4	算符作为参数	158
5.6.5	算符递归	160
5.6.6	算符使用策略	161
5.6.7	小结	161
	本章小结	161
	习题 5	162
第 6 章	类与对象	164
6.1	对象模型	164
6.2	类实体	165
6.3	模块	166
6.4	创建和访问对象	166
6.5	接口对象类型	167
6.6	多重实现	167
6.7	包容多态性	168
6.8	support 类型扩展	168
6.9	object 超类型	169
6.10	继承	169
6.11	对象体系的其他特点	171
6.12	Visual Prolog 5 与 Visual Prolog 6 的差异	171
6.12.1	句点	171
6.12.2	谓词	171
6.12.3	谓词论域	172
6.12.4	引用论域	172
6.12.5	函数子句	172
6.12.6	常量	173
6.12.7	事实	173
6.12.8	事实变量	173
6.12.9	嵌套表达式与函数	174
6.12.10	编译器命令	174
6.12.11	条件编译	174

6.12.12 输入输出及特殊论域	175
6.12.13 省略与匿名参数类型	175
6.12.14 对象与类	176
6.12.15 库支持	176
本章小结	180
习题 6	180
 第 7 章 Visual Prolog 编程	182
7.1 Visual Prolog 基础	182
7.1.1 程序结构	182
7.1.2 目标	184
7.1.3 文件考虑	185
7.1.4 作用域访问	185
7.1.5 面向对象	186
7.1.6 一个完整的例子: family1.prj6	186
7.1.7 程序的取舍	192
7.1.8 小结	193
7.2 Visual Prolog 的 GUI 编程	193
7.2.1 GUI 概述	194
7.2.2 GUI 对事件的响应	195
7.2.3 开始一个 GUI 项目	195
7.2.4 创建模态对话框	196
7.2.5 修改菜单	200
7.2.6 修改工具栏	202
7.2.7 在程序中添加主代码	204
7.2.8 压缩相关代码	206
7.2.9 分析所做的工作	209
7.2.10 运行程序	211
7.2.11 小结	212
7.3 Visual Prolog 的逻辑层	212
7.3.1 初始准备阶段	212
7.3.2 创建业务逻辑层	212
7.3.3 在业务逻辑层上工作	213
7.3.4 创建业务逻辑类	214
7.3.5 理解业务逻辑类	217
7.3.6 连接业务逻辑层到 GUI	217
7.3.7 理解事件处理程序	220
7.3.8 运行代码	221
7.3.9 细节考虑	221

7.3.10 小结	221
7.4 Visual Prolog 的数据层	222
7.4.1 基本概念	222
7.4.2 程序	222
7.4.3 非模态对话框	225
7.4.4 FamilyData 包	226
7.4.5 接口	226
7.4.6 FamilyDL 包	227
7.4.7 FamilyDL 包的代码	228
7.4.8 FamilyBLL 包的代码	229
7.4.9 数据层的特征	230
7.4.10 小结	231
本章小结	231
习题 7	231
 第 8 章 编写 CGI 程序	232
8.1 概述	232
8.2 编写 CGI 程序基础	232
8.2.1 公共网关接口	232
8.2.2 CGI 程序	235
8.2.3 测试 CGI 程序	238
8.2.4 用 Visual Prolog 6 创建 CGI 程序	239
8.2.5 测试 example1	241
8.2.6 应用程序功能分析	242
8.2.7 输入流分析	242
8.3 编写实用的 CGI 应用程序	242
8.3.1 将信息从 HTML 文件传输至 CGI 程序	242
8.3.2 解释信息流的高级 CGI 应用程序	244
8.3.3 信息从网络服务器到浏览器的传输	246
8.3.4 CGI 应用程序简评	247
8.3.5 取代 CGI 程序的候选方案	247
8.3.6 加速 CGI 应用程序	247
8.3.7 CGI 程序的客户端	247
8.3.8 使用 Javascript 对象的高级 CGI 应用程序	249
8.3.9 安全性问题	251
8.3.10 防止 CGI 程序被盗链	252
8.3.11 小结	253
8.4 CGI 应用程序测试实例	253
8.4.1 安装 TinyWeb 网络服务器	253

8.4.2	TinyWeb 的根目录	254
8.4.3	TinyWeb 的端口	254
8.4.4	调试例子程序	254
8.4.5	用其他网络服务器运行例子程序	254
	本章小结	254
	习题 8	255
第 9 章	编码风格	256
9.1	基本元素	256
9.1.1	关键字	256
9.1.2	半关键字	256
9.1.3	文字	257
9.1.4	标识符	257
9.1.5	常量	257
9.1.6	变量	257
9.1.7	谓词	257
9.1.8	论域	258
9.1.9	类和接口	258
9.2	推荐格式	258
9.2.1	折行	258
9.2.2	缩排	259
9.2.3	对齐	259
9.2.4	空格字符	259
9.3	程序结构	259
9.3.1	段	259
9.3.2	类、接口及实现	260
9.3.3	谓词声明	260
9.3.4	论域	260
9.3.5	子句	261
9.3.6	不确定性循环	261
9.3.7	Word 格式化代码	261
9.4	程序设计语用学	262
9.4.1	常规技巧	262
9.4.2	布尔值	263
9.4.3	截断	263
9.4.4	红色截断和绿色截断	264
9.4.5	指派输入格式	265
9.4.6	异常和错误处理	266
9.4.7	内部错误和其他错误	266

9.5 存储管理	267
9.5.1 存储器	267
9.5.2 运行堆栈	267
9.5.3 尾部调用优化	268
9.5.4 运行栈耗尽	268
9.5.5 全局栈	268
9.5.6 G-堆栈耗尽	268
9.5.7 堆和垃圾回收	269
9.5.8 垃圾回收	269
9.5.9 Finalizers	270
9.5.10 数据在什么地方分配	270
9.5.11 堆中数据	270
9.5.12 多线程和存储	271
9.6 异常处理	271
9.6.1 如何捕获异常	271
9.6.2 如何构造自己的异常	273
9.6.3 如何继续另一个异常	274
本章小结	276
习题 9	276

第 3 部分 语言参考

第 10 章 Visual Prolog 语言元素	280
10.1 类型	280
10.2 对象系统	281
10.2.1 外部视图	281
10.2.2 内部视图	282
10.3 作用域和可视性	283
10.3.1 名字分类	283
10.3.2 可视性、隐蔽性及限定性	285
10.4 词法结构	286
10.4.1 程序单元	286
10.4.2 标记	287
10.4.3 文字	289
本章小结	291
习题 10	291
第 11 章 Visual Prolog 数据元素	292
11.1 论域段	292

11.1.1	类型名	292
11.1.2	复合论域	293
11.1.3	列表论域	295
11.1.4	引用论域	296
11.1.5	谓词论域	296
11.1.6	整型论域	301
11.1.7	实型论域	302
11.2	通用类型和根类型	303
11.2.1	通用类型	303
11.2.2	根类型	304
	本章小结	304
	习题 11	304
第 12 章	Visual Prolog 程序元素	305
12.1	项	305
12.1.1	项的基本概念	305
12.1.2	运算符	306
12.1.3	类成员访问	307
12.1.4	对象成员访问	308
12.1.5	全局实体的访问	308
12.1.6	论域、算符和常量访问	308
12.2	常量	308
12.2.1	常量段	308
12.2.2	常量定义	309
12.3	谓词	309
12.3.1	谓词段	309
12.3.2	构造段	310
12.3.3	接口谓词	311
12.3.4	变元	313
12.4	子句	313
12.4.1	子句段	313
12.4.2	目标段	314
12.5	事实	315
12.5.1	事实段	315
12.5.2	事实声明	315
12.5.3	事实变量	316
12.5.4	事实	316
12.6	评估	317
12.6.1	回溯	317

12.6.2	谓词调用	317
12.6.3	合一	318
12.6.4	引用论域	319
12.6.5	匹配	319
12.6.6	嵌套的函数调用	320
12.6.7	变量与常量	320
12.6.8	算术表达式	321
12.6.9	事实断言与撤销	322
12.6.10	失败谓词和成功谓词	322
12.6.11	逻辑与	322
12.6.12	逻辑或	322
12.6.13	逻辑非	323
12.6.14	截断	323
12.6.15	谓词 finally/2	324
12.7	程序段	325
	本章小结	325
	习题 12	326
第 13 章	编译单元	327
13.1	接口	327
13.1.1	接口的基本概念	327
13.1.2	接口与对象	328
13.1.3	开放限定	328
13.1.4	支持限定	329
13.2	类声明	330
13.3	类实现	332
13.3.1	类实现的基本概念	332
13.3.2	继承限定	334
13.3.3	归结限定	334
13.3.4	委托限定	337
13.3.5	This 修饰	339
13.3.6	构造器	341
13.3.7	终结	344
13.4	类型转换	345
13.4.1	隐式转换	345
13.4.2	显式转换	346
13.5	条件编译	349
13.6	异常处理	349
13.7	预处理程序指令	350

13.7.1	条件编译指令	350
13.7.2	源文件包含	351
13.7.3	编译时间信息	351
	本章小结	353
	习题 13	354
第 14 章	内部论域、谓词和常量	355
14.1	概述	355
14.2	内部常量详解	357
14.3	内部论域详解	358
14.4	内部谓词详解	363
	本章小结	380
	习题 14	380
第 15 章	与其他编程语言接口	381
15.1	外部代码	381
15.2	关键问题	381
15.3	调用约定和链接名	382
15.4	数据表示	383
15.4.1	举例	383
15.4.2	外部链接库	384
15.5	存储管理	385
15.5.1	典型解决方案	385
15.5.2	垃圾收集和全局堆栈	386
15.6	Win32 API 函数	386
	本章小结	388
	习题 15	388
附录	术语表	389
A	389
B	390
C	391
D	394
E	396
F	397
G	398
H	399
I	399
K	401

L.....	401
M.....	402
N.....	403
O.....	404
P.....	405
R.....	407
S.....	408
T.....	411
U.....	412
V.....	413
参考文献.....	414

第 1 部分 基础知识

第 1 章 人工智能概述

第 2 章 知识表示方法

第 3 章 AI 编程基础

第 1 章 人工智能概述

人工智能（artificial intelligence, AI）是计算机科学、控制论、信息论、神经生理学、心理学、语言学等诸多学科相互交叉、相互渗透而发展起来的一门新兴边缘学科。它主要研究如何用机器（计算机）来模拟和实现人类的智能行为。人工智能技术同原子能技术、空间技术一起被称为 20 世纪三大科技成就。人工智能中的专家系统、机器学习、自然语言理解等分支领域已经投入使用。一个智能化信息处理的新时代正向世界走来。近年来，计算机网络，特别是因特网的迅猛发展和广泛应用，又为人工智能提供了新的广阔天地。信息化需要智能化的支持，人工智能在信息高速公路上也将发挥重要作用。目前，世界各国对人工智能的研究都十分重视，纷纷投入大量的人力、物力和财力，激烈争夺这一高新技术的制高点。

人工智能的前景诱人，同时也任重道远。本章作为概述，主要讨论人工智能的定义、研究目标、研究内容、研究途径与方法、主要特点、研究领域、基本技术、形成过程及发展趋势等，目的在于展示一个处于不断发展中的人工智能的概貌。

1.1 人工智能的概念

1.1.1 人工智能

所谓“人工智能”是指用计算机模拟或实现的智能。作为一门学科，人工智能研究的是如何使机器（计算机）具有智能的科学和技术，特别是人类智能如何在计算机上实现或再现的科学和技术。因此，从学科角度讲，当前的人工智能是计算机科学的一个分支。

人工智能虽然是计算机科学的一个分支，但它的研究却不仅涉及计算机科学，而且还涉及脑科学、神经生理学、心理学、语言学、逻辑学、认知（思维）科学、行为科学和数学、信息论、控制论和系统论等众多学科领域。因此，人工智能实际上是一门综合性的交叉学科和边缘学科。

广义的人工智能学科是模拟、延伸和扩展人的智能，研究与开发各种机器智能和智能机器的理论、方法与技术的综合性学科。

人工智能是一个含义很广的词语，在其发展过程中，具有不同学科背景的人工智能学者对它有着不同的理解，提出了一些不同的观点，人们称这些观点为符号主义(symbolism)、连接主义(connectionism)和行为主义(actionism)等，或者叫做逻辑学派(logicism)、仿生学派(bionicsism)和生理学派(physiologism)。此外还有计算机学派、心理学派和语言学派等。

斯坦福大学人工智能研究中心的尼尔逊(N. J. Nilsson)教授从处理的对象出发，认为“人工智能是关于知识的科学，即怎样表示知识、怎样获取知识和怎样使用知识的科学”。

麻省理工学院温斯顿（P. H. Winston）教授则认为“人工智能就是研究如何使计算机去做过去只有人才能做的富有智能的工作”。斯坦福大学费根鲍姆（E.A. Feigenbaum）教授从知识工程的角度出发，认为“人工智能是一个知识信息处理系统”。

综合各种不同的人工智能观点，可以从“能力”和“学科”两个方面对人工智能进行定义。从能力的角度来看，人工智能是相对于人的自然智能而言的，所谓人工智能是指用人工的方法在机器（计算机）上实现的智能；从学科的角度来看，人工智能是作为一个学科名称来使用的，所谓人工智能是一门研究如何构造智能机器或智能系统，使它能模拟、延伸和扩展人类智能的学科。总之，人工智能是一门综合性的边缘学科。它借助于计算机建造智能系统，完成诸如模式识别、自然语言理解、程序自动设计、自动定理证明、机器人、专家系统等智能活动。它的最终目标是构造智能机。

如何衡量机器是否具有智能呢？早在 1950 年，人工智能还没有作为一门学科正式出现之前，英国数学家图灵（A. M. Turing）就在他发表的一篇文章“Computing Machinery and Intelligence（计算机与智能）”中提出了“机器能思维”的观点，并设计了一个很著名的测试机器智能的实验，称为“图灵测试”或“图灵实验”。该测试的参加者由一位测试主持人和两个被测试者组成。要求两个被测试者中的一个 是人，另一个是机器。测试规则是：让测试主持人和每个被测试者分别位于彼此不能看见的房间中，相互之间只能通过计算机终端进行会话。测试开始后，由测试主持人向被测试者提出各种具有智能性的问题，但不能询问测试者的物理特征。被测试者在回答问题时，都应尽量使测试者相信自己是“人”，而另一位是“机器”。在这个前提下，要求测试主持人区分被测试者哪个是人，哪个是机器。如果无论如何更换测试主持人和被测试者中的人，测试主持人能分辨出人和机器的概率都小于 50%，则认为该机器具有了智能。作为人的一方不能判定对方是人还是机器，那么就认为对方的那台机器达到了人的智能。

对图灵的这个测试标准，也有人提出了疑义：认为该测试仅反映了结果的比较，既没有涉及思维的过程，也没有明确参加实验的人是小孩还是具有良好素质的成年人。尽管如此，它对人工智能这门学科的发展所产生的影响则是十分深远的。

要研究人工智能，当然要涉及什么是智能的问题，但这却是一个难以准确回答的问题。因为关于智能，至今还没有一个确切的公认的定义。这是由于智能是脑，特别是人脑的属性或者说产物。但人脑的奥秘至今还未完全揭开。从系统的观点看，人脑是一个复杂的、开放的、动态的巨系统。它的内部结构和工作机理，至今人们还不完全清楚。所以，这就导致了对于智能的多种说法。譬如有人说智能的基础是知识（因为没有知识的智能是不可想像的）；有人说智能的关键是思维（因为知识是思维产生的）；有人说智能取决于感知和行为，认为智能是在系统与周围环境不断“刺激—反应”的交互中发展和进化的。作者认为，从内涵来讲，智能应该是知识+思维；从外延来讲，智能就是发现规律、运用规律的能力（或者说获取知识、运用知识的能力）和分析问题、解决问题的能力。

1.1.2 为什么要研究人工智能

电子计算机是迄今为止最有效的信息处理工具，以至于人们称它为“电脑”。但现在的普通计算机系统的智能还相当低下，譬如缺乏自适应、自学习、自优化等能力，也缺乏

社会常识或专业知识等，而只能是被动地按照人们为它事先安排好的工作步骤进行工作。因而它的功能和作用就受到很大的限制，难以满足越来越复杂和越来越广泛的社会需求。既然计算机和人脑一样都可进行信息处理，那么是否也能让计算机同人脑一样也具有智能呢？这正是人们研究人工智能的初衷。

事实上，如果计算机自身也具有一定智能的话，那么它的功效将会发生质的飞跃，成为名副其实的“电脑”。这样的电脑将是人脑更为有效的扩展和延伸，也是人类智能的扩展和延伸，其作用将是不可估量的。例如，用这样的电脑武装起来的机器人就是智能机器人。智能机器人的出现，将标志着人类社会进入了一个新的时代。

研究人工智能也是当前信息化社会的迫切要求。人类社会现在已经进入了信息化时代，但信息化的进一步发展，就必须有智能技术的支持。例如，当前迅速发展着的因特网就强烈地需要智能技术。特别是当人们要在因特网上构筑信息高速公路时，其中许多技术问题就要用人工智能的方法来解决。这就是说，人工智能技术在因特网和未来的信息高速公路上将发挥重要作用。

智能化也是自动化发展的必然趋势。自动化发展到一定水平，再向前发展就是智能化，即智能化是继机械化、自动化之后，人类生产和生活中的又一个技术特征。

另外，研究人工智能，对探索人类自身智能的奥秘也可提供有益的帮助。因为人们可以通过电脑对人脑进行模拟，从而揭示人脑的工作原理，发现自然智能的渊源。

1.1.3 人类智能的计算机模拟

人类的认知过程是个非常复杂的行为，至今仍未能被完全解释。人们从不同的角度对它进行研究，从而形成诸如认知生理学、认知心理学和认知工程学等相关学科。对这些学科的深入研究已超出本书范围。这里仅讨论几个与人工智能关系密切的问题。

1. 研究认知过程的任务

人的心理活动具有不同的层次，它可与计算机的层次相比较，如图 1.1 所示。心理活动的最高层级是思维策略，中间一层是初级信息处理，最低层级为生理过程，即中枢神经系统、神经元和大脑的活动。与此相应的是计算机的程序、语言和硬件。

研究认知过程的主要任务是探求高层次思维决策与初级信息处理的关系，并用计算机程序来模拟人的思维策略水平，而用计算机语言模拟人的初级信息处理过程。

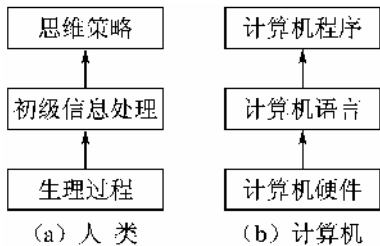


图 1.1 人类认知活动与计算机的比较

令 T 表示时间变量， x 表示认知操作 (cognitive operation)， x 的变化 Δx 为当时机体状态 S (机体的生理和心理状态以及脑子里的记忆等) 和外界刺激 R 的函数。当外界刺激作用到处于某一特定状态的机体时，便发生变化，即语言和硬件。

计算机也以类似的原理进行工作。在规定时间内，计算机存储的记忆相当于机体的状态；计算机的输入相当于机体施加的某种刺激。在得到输入后，计算机便进行操作，

使得其内部状态随时间发生变化。可以从不同的层次来研究这种计算机系统。这种系统以人的思维方式为模型进行智能信息处理 (intelligent information processing)。显然, 这是一种智能计算机系统。设计适用于特定领域的这种高水平智能信息处理系统 (也称为专家系统) 是研究认知过程的一个具体而又重要的目标。例如, 一个具有智能信息处理能力的自动控制系统就是一个智能控制系统, 它可以是专家控制系统, 或者是智能决策系统等。

2. 智能信息处理系统的假设

可以把人看成一个智能信息处理系统。信息处理系统又叫物理符号系统 (physical symbol system)。所谓符号就是模式 (pattern)。任一模式, 只要它能与其他模式相区别, 它就是一个符号。不同的汉语拼音字母或英文字母就是不同的符号。对符号进行操作就是对符号进行比较, 从中找出相同的和不同的符号。物理符号系统的基本任务和功能就是辨认相同的符号和区别不同的符号。为此, 这种系统就必须能够辨别出不同符号之间的实质差别。符号既可以是物理符号, 也可以是头脑中的抽象符号, 或者是电子计算机中的电子运动模式, 还可以是头脑中神经元的某些运动方式。一个完善的符号系统应具有下列 6 种基本功能:

- (1) 输入符号 (input);
- (2) 输出符号 (output);
- (3) 存储符号 (store);
- (4) 复制符号 (copy);
- (5) 建立符号结构: 通过找出各符号间的关系, 在符号系统中形成符号结构;
- (6) 条件性迁移 (conditional transfer): 根据已有符号, 继续完成活动过程。

如果一个物理符号系统具有上述全部 6 种功能, 能够完成这个全过程, 那么它就是一个完整的物理符号系统。人能够输入信号, 如用眼睛看, 用耳朵听, 用手触摸等。计算机也能通过卡片或纸带打孔、磁带或键盘打字等方式输入符号。人具有上述 6 种功能, 现代计算机也具备物理符号系统的这 6 种功能。

假设 任何一个系统, 如果它能表现出智能, 那么它就必定能够执行上述 6 种功能。反之, 任何系统如果具有这 6 种功能, 那么它就能够表现出智能。这种智能指的是人类所具有的那种智能。把这个假设称为物理符号系统的假设。

物理符号系统的假设伴随有 3 个推论, 或称为附带条件。

推论一 既然人具有智能, 那么他 (她) 就一定是一个物理符号系统。

人之所以能够表现出智能, 就是基于他的信息处理过程。

推论二 既然计算机是一个物理符号系统, 它就一定能够表现出智能的基本条件。这是人工智能的基本条件。

推论三 既然人是一个物理符号系统, 计算机也是一个物理符号系统, 那么就能够用计算机来模拟人的活动。

值得指出, 推论三并不一定是从推论一和推论二推导出来的必然结果。因为人是物理符号系统, 具有智能; 计算机也是一个物理符号系统, 也具有智能, 但他 (它) 们可以用不同的原理和方式进行活动。所以计算机并不一定都是模拟人活动的, 它可以编制出一些

复杂的程序来求解方程式，进行复杂的计算。不过，计算机的这种运算过程未必就是人类的思维过程。

可以按照人类的思维过程来编制计算机程序，这项工作就是人工智能的研究内容，也是智能控制的主要研究内容。如果做到了这一点，就可以用计算机在形式上来描述人的思维活动过程，或者建立一个理论来说明人的智力活动过程。

3. 人类智能的计算机模拟

帕梅拉·麦考达克（Pamela McCorduck）在她的著名的人工智能历史研究《机器思维》（*Machine Who Think*, 1979）中曾经指出：在复杂的机械装置与智能之间存在着长期的联系。从几世纪前出现的神话般的复杂巨钟和机械自动机开始，人们已对机器操作的复杂性与自身的智能活动进行直接联系。今天，新技术已使人们所建造的机器的复杂性大为提高。现代电子计算机要比以往的任何机器复杂几十倍、几百倍，甚至几千倍以上。

计算机的早期工作主要集中在数值计算方面。然而人类最主要的智力活动并不是数值计算，而是在逻辑推理方面。物理符号系统假设的推论一也告诉我们，人有智能，所以他是一个物理符号系统。推论三指出，可以编写出计算机程序去模拟人类的思维活动。这就是说，人和计算机这两个物理符号系统所使用的物理符号是相同的，因而计算机可以模拟人类的智能活动过程。计算机的确能够很好地执行许多智能功能，如下棋、证明定理、翻译语言文字和解决难题等。这些任务是通过编写与执行模拟人类智能的计算机程序来完成的。当然，这些程序只能接近于人的行为，而不可能与人的行为完全相同。此外，这些程序所能模拟的智能问题，其水平还是很有限的。

作为例子，考虑下棋的计算机程序。现有的国际象棋程序是十分熟练的、具有人类专家棋手水平的最好实验系统，但是下得没有像人类国际象棋大师那样好。该计算机程序对每个可能的走步空间进行搜索，它能够同时搜索几千种走步，进行有效搜索的技术是人工智能的核心思想之一。不过，计算机不一定是最好的棋手，其原因在于：向前看并不是下棋所必须具有的一切，需要彻底搜索的走步又太多；在寻找和估计替换走步时并不能确信能够导致博弈的胜利。国际象棋大师们具有尚不能解释的能力。一些心理学家指出，当象棋大师们盯着一个棋位时，在他们的脑子里出现了几千盘重要的棋局，这大概能够帮助他们决定最好的走步。

近年来，智能计算机的研究取得许多重大进展。对神经型智能计算机的研究就是一个新的范例，它必将为模拟人类智能做出新的贡献。

神经计算机（neural computer）能够以类似人类的方式进行“思考”，它力图重建人脑的形象。据日本通产省（MITI）报导，对神经计算机系统的可行性研究早于1989年4月底完成，并提出了该系统的长期研究计划的细节。在美国、英国、中国和其他一些国家，都有众多的研究小组投入对“神经网络”的研究。据预测，神经计算机在本世纪将进入实用阶段，并将有产品投放市场。

人脑这个神奇的器官能够复制大量的交互作用，快速处理极其大量的信息，同时执行几项任务。迄今为止的所有计算机，基本上都未能摆脱冯·诺依曼机的体系结构，只能依次对单个问题进行“求解”。即使是20世纪80年代初期的并行处理计算机，其运行性能仍然十分有限。人们期望，对神经计算（neural computing）的研究将造出神经计算机，大大

提高信息处理能力，达到更高的人工智能水平。

1.2 人工智能的研究目标

关于人工智能的研究目标，目前还没有一个统一的说法。从研究的内容出发，李文特和费根鲍姆提出了人工智能的9个最终目标。

1. 理解人类的认识

此目标研究人类如何进行思维，而不是研究机器如何工作。要尽量深入了解人的记忆、问题求解能力、学习的能力和一般的决策等过程。

2. 有效的自动化

此目标是在需要智能的各种任务上用机器取代人，其结果是要建造执行起来和人一样好的程序。

3. 有效的智能拓展

此目标是建造思维上的弥补物，有助于人们的思维更富有成效、更快、更深刻、更清晰。

4. 超人的智力

此目标是建造超过人的性能的程序。如果越过这一知识阈值，就可以导致进一步地增殖，如制造行业上的革新、理论上的突破、超人的教师和非凡的研究人员等。

5. 通用问题求解

此目标的研究可以使程序能够解决或至少能够尝试其范围之外的一系列问题，包括过去从未听说过的领域。

6. 连贯性交谈

此目标类似于图灵测试，它可以令人满意地与人交谈。交谈使用完整的句子，而句子是用某一种人类的语言。

7. 自治

此目标是一系统，它能够主动地在现实世界中完成任务。它与下列情况形成对比：仅在某一抽象的空间做规划，在一个模拟世界中执行，建议人去做某种事情。该目标的思想是：现实世界永远比人们的模型要复杂得多，因此它才成为测试所谓智能程序的惟一公正的手段。

8. 学习

此目标是建造一个程序，它能够选择收集什么数据和如何收集数据。然后再进行数据的收集工作。学习是将经验进行概括，成为有用的观念、方法、启发性知识，并能以类似

方式进行推理。

9. 存储信息

此目标就是要存储大量的知识，系统要有一个类似于百科词典式的，包含广泛范围知识的知识库。

要实现这些目标，需要同时开展对智能机理和智能构造技术的研究。即使对图灵所期望的那种智能机器，尽管它没有提到思维过程，但要真正实现这种智能机器，却同样离不开对智能机理的研究。因此，揭示人类智能的根本机理，用智能机器去模拟、延伸和扩展人类智能应该是人工智能研究的根本目标，或者叫远期目标。

人工智能研究的远期目标是要制造智能机器。具体来讲，就是要使计算机具有看、听、说、写等感知能力和交互功能，具有联想、推理、理解、学习等高级思维能力，还要有分析问题、解决问题和发明创造的能力。简言之，也就是使计算机像人一样具有自动发现规律和利用规律的能力，或者说具有自动获取知识和利用知识的能力，从而扩展和延伸人的智能。

人工智能的远期目标涉及脑科学、认知科学、计算机科学、系统科学、控制论及微电子等多种学科，并有赖于这些学科的共同发展。但从目前这些学科的现状来看，实现人工智能的远期目标还需要有一个较长的时期。

人工智能研究的近期目标是实现机器智能，是研究如何使现有的计算机更聪明，即先部分地或某种程度地实现机器的智能，从而使现有的计算机更灵活、更好用和更有用，成为人类的智能化信息处理工具，使它能够运用知识去处理问题，能够模拟人类的智能行为，如推理、思考、分析、决策、预测、理解、规划、设计和学习等。为了实现这一目标，人们需要根据现有计算机的特点，研究实现智能的有关理论、方法和技术，建立相应的智能系统。

实际上，人工智能的远期目标与近期目标是相互依存的。远期目标为近期目标指明了方向，而近期目标则为远期目标奠定了理论和技术基础。同时，近期目标和远期目标之间并无严格界限，近期目标会随人工智能研究的发展而变化，并最终达到远期目标。

需指出的是，人工智能的远期目标虽然现在还不能全部实现，但在某些侧面，当前的机器智能已表现出相当高的水平。例如，在机器博弈、机器证明、识别和控制等方面，当前的机器智能的确已达到或接近了能同人类抗衡和媲美的水平。下面的两例可见一斑：1995年，美国研制的自动汽车（即智能机器人驾驶的汽车），在高速公路上以55km/h的速度，从美国的东部一直开到西部，其中的98.8%的操作都是由机器自动完成的。1997年5月3日至11日，IBM公司的深蓝巨型计算机与蝉联12年之久的世界象棋冠军卡斯帕罗夫进行了6场比赛，厮杀得难分难解。在决定胜负的最后一局比赛中，深蓝以不到1h的时间，在第19步棋就轻易逼得卡斯帕罗夫俯首称臣，从而以3.5分比2.5分的总成绩取得胜利。

总之，无论是人工智能研究的近期目标，还是远期目标，摆在人们面前的任务异常艰巨，还有一段很长的路要走。在人工智能的基础理论和物理实现上，还有许多问题要解决。当然，仅仅只靠人工智能工作者是远远不行的，还应该聚集诸如心理学家、逻辑学家、数学家、哲学家、生物学家和计算机科学家等，依靠群体的共同努力，去实现人类梦想的“第2次知识革命”。

1.3 人工智能研究的基本内容及特点

本节介绍人工智能研究的基本内容、人工智能研究的途径和方法，以及人工智能研究的主要特点。

1.3.1 人工智能研究的基本内容

关于人工智能的研究内容，各种不同学派、不同研究领域以及人工智能发展的不同时期，对其有着一些不同的看法。下面根据人工智能的现状，给出几个对实现人工智能系统来说具有一般意义的基本内容。

1. 认知建模

所谓认知可以一般地认为是和情感、动机、意志相对应的理智或认识过程，或者说是为了一定目的，在一定的心理结构中进行的信息加工过程。

美国心理学家浩斯顿（Houston）等人曾把对认知（cognition）的看法归纳为以下 5 种主要类型：

- （1）认知是信息的处理过程；
- （2）认知是心理上的符号运算；
- （3）认知是问题求解；
- （4）认知是思维；

（5）认知是一组相关的活动，如知觉、记忆、思维、判断、推理、问题求解、学习、想像、概念形成及语言使用等。

实际上人类的认知过程是非常复杂的，人们对其研究形成了认知科学（也称思维科学）。因此，认知科学是研究人类感知和思维信息处理过程的一门学科，它包括从感觉的输入到复杂问题的求解，从人类个体智能到人类社会智能的活动，以及人类智能和机器智能的性质。其主要研究目的就是要说明和解释人类在完成认知活动时是如何进行信息加工的。

认知科学是人工智能的重要理论基础，对人工智能发展起着根本性的作用。认知科学涉及的问题非常广泛，除了像浩斯顿提出的知觉、语言、学习、记忆、思维、问题求解、创造、注意、想像等相关联活动外，还会受到环境、社会、文化背景等方面的影响。从认知观点看，人工智能不能仅限于逻辑思维的研究，还必须深入开展对形象思维和灵感思维的研究。只有这样，才能使人工智能具有更坚实的理论基础，才能为智能计算机系统的研制提供更新的思想，创造更新的途径。

2. 机器感知

所谓机器感知就是要让计算机具有类似于人的感知能力，如视觉、听觉、触觉、嗅觉、味觉。在这些感知能力中，目前研究较多、较为成功的是机器视觉（或叫计算机视觉）和机器听觉（或叫计算机听觉）。计算机视觉就是给计算机配上能看的视觉器官，如摄像机

等，使它可以识别并理解文字、图像、景物等；计算机听觉就是给计算配上能听的听觉器官，如话筒等，使计算机能够识别并理解语言、声音等。

机器感知是计算机智能系统获取外部信息的最主要途径，也是机器智能不可缺少的重要组成部分。对计算机视觉与听觉的研究，目前已在人工智能中形成了一些专门的研究领域，如计算机视觉、模式识别、自然语言理解等。

3. 机器思维

所谓机器思维就是让计算机能够对感知到的外界信息和自己产生的内部信息进行思维性加工。由于人类的思维功能包括逻辑思维、形象思维和灵感思维，因此机器思维的研究也应该包括这几个方面。为了实现机器的思维功能，需要在知识的表示、组织及推理方法，各种启发式搜索及控制策略，神经网络、人脑结构及其工作原理等方面进行研究。

由于人类智能主要来自于大脑的思维活动，因此机器智能也主要应该通过机器的思维功能来实现。机器思维是机器智能的重要组成部分。

4. 机器学习

所谓机器学习就是让计算机能够像人那样自动地获取新知识，并在实践中不断地完善自我和增强能力。机器学习是机器具有智能的根本途径，也是人工智能研究的核心问题之一。目前，人类根据对学习的已有认识，已经研究出了不少机器学习方法，如机械学习、类比学习、归纳学习、发现学习、遗传学习和连接学习等。

5. 机器行为

所谓机器行为就是让计算机能够具有像人那样的行动和表达能力，如走、跑、拿、说、唱、写、画等。如果把机器感知看作智能系统的输入部分，那么机器行为则可看作智能系统的输出部分。机器人学作为人工智能的一个研究领域，包含了机器行为方面的研究。

6. 智能系统与智能计算机

无论是人工智能的近期目标还是远期目标，都需要建立智能系统或构造智能机器，因此需要开展对系统模型、构造技术、构造工具及语言环境等方面的研究。

1.3.2 人工智能的研究途径与方法

智能是脑特别是人脑所具有的。那么要实现人工智能，自然就离不开对人脑的借鉴，其中包括对人脑的结构、功能和人脑具有智能的原因、过程等的借鉴。于是，就产生了如下几种人工智能研究途径和方法。

1. 基于结构模拟的神经计算

所谓结构模拟就是根据人脑的生理结构和工作机理实现计算机的智能，即人工智能。人脑的生理结构是由大量神经细胞组成的神经网络。由于这个网络太庞大、太复杂——研究表明，人脑是由大约 10^{11} 个神经细胞组成的一个动态的、开放的、高度复杂的巨系统，

以至于人们至今对它的生理结构和工作机理还未完全弄清楚。因此，对人脑的真正和完全模拟，一时还难以办到。所以，目前的结构模拟只是对人脑的局部或近似模拟。具体来讲，就是用人工神经元（神经细胞）组成的人工神经网络来作为信息和知识的载体，用所谓神经计算的方法实现学习、联想、识别和推理等功能，从而来模拟人脑的智能行为，使计算机表现出某种智能。

所以结构模拟法也就是基于人脑的生理模型，采用数值计算的方法，从微观上来模拟人脑，实现机器智能。这种方法一般是通过神经网络的“自学习”获得知识，再利用知识解决问题。神经网络具有高度的并行分布性、很强的鲁棒性和容错性。所以，它擅长模拟人脑的形象思维，便于实现人脑的低级感知功能，例如，图像、声音信息的识别和处理。

采用结构模拟，运用神经网络和神经计算的方法研究人工智能者，被称为生理学派、连接主义。这种方法早在20世纪40年代就已出现，但由于种种原因而发展缓慢，甚至一度出现低潮，直到20世纪80年代中期才重新崛起。当前已成为人工智能一个非常热门的研究方向。

2. 基于功能模拟的符号推演

由于人脑的奥秘至今还未彻底揭开，所以人们就在当前的数字计算机上，对人脑从功能上进行模拟，实现人工智能。这种途径称为功能模拟法。

具体来讲，功能模拟法就是以人脑的心理模型，将问题或知识表示成某种逻辑网络，采用符号推演的方法，实现搜索、推理、学习等功能，从宏观上来模拟人脑的思维，实现机器智能。

基于功能模拟的符号推演，是人工智能研究中最早使用且直至目前还主要使用的方法。人工智能的许多重要成果也都是用该方法取得的，如自动推理、定理证明、专家系统、机器博弈等等。这种方法一般是利用显式的知识（库）和推理（机）来解决问题的。所以，它擅长模拟人脑的逻辑思维，便于实现人脑的高级认知功能，如推理、决策等。以功能模拟和符号推演研究人工智能者，被称为心理学派、逻辑学派、符号主义。

需说明的是，人们使用功能模拟方法的原因，一方面是由于至今人们对大脑的生理结构和工作机理还没有完全弄清楚，另一方面是由于如下原因：

- （1）当前的数字计算机可以方便地实现高速的符号处理；
- （2）这种方法可以显式地表示知识，容易表达人的心理模型；
- （3）智能行为也并非仅神经网络那样的结构形式所独有。

以上两种方法，是当前人工智能研究的两条主要途径。它们各有所长，也各有所短。从这两种方法所擅长处理的问题来看，它们都有一定的局限性，而且刚好互为补充。因此，至少从目前来看，这两种研究途径并不是互相取代，而是并存和互补的关系。事实上，功能模拟虽然仅是对大脑的功能模拟，但它对揭示大脑生理奥秘仍有许多借鉴之处，结构模拟虽然主观上是要对大脑实现仿真，但由于至今人们对大脑的工作原理还没有完全搞清楚，因而也带有一定程度的功能模拟性。再从当前的研究现状来看，人们将模糊推理与神经计算相结合，已展现出相得益彰的喜人前景。因此，将功能模拟与结构模拟相结合是当前人工智能研究的总趋势。

3. 基于行为模拟的控制进化

除了上述两种研究途径和方法外，还有一种基于“感知—行为”模型的研究途径和方法，称其为行为模拟法。这种方法是模拟人在控制过程中的智能活动和行为特性，如自寻优、自适应、自学习、自组织等，来研究和实现人工智能。基于这一方法研究人工智能的典型代表要算 MIT 的 R. Brooks 教授，它研制的六足行走机器人（也称为人造昆虫或机器虫），曾引起人工智能界的轰动。这个机器虫可以看作是新一代的“控制论动物”，它具有一定的自适应能力，是一个运用行为模拟，即控制进化方法研究人工智能的代表作。

事实上，Brooks 教授的工作代表了称为“现场（situated）AI”的人工智能新方向。现场 AI 强调智能系统与环境的交互，认为智能取决于感知和行动，智能行为可以不需要知识，提出“没有表示的智能”、“没有推理的智能”的观点，主张智能行为的“感知—动作”模式，认为人的智能、机器智能可以逐步进化，但只能在现实世界中与周围环境的交互中体现出来。智能只能放在环境中才是真正的智能，智能的高低主要表现在对环境的适应性上。

以行为模拟方法研究人工智能者，被称为行为主义、进化主义、控制论学派。行为主义曾强烈地批评传统的人工智能（主要指符号主义，也涉及连接主义）对真实世界的客观事物和复杂境遇做了虚假的、过分简化的抽象。

1.3.3 人工智能研究的主要特点

目前的计算机系统仍未彻底突破传统的冯·诺依曼结构，这种二进制表示的集中串行工作方式具有较强的逻辑运算功能和很快的算术运算速度，但与人脑的组织结构和思维功能有很大差别。研究表明，人脑大约有 10^{11} 个神经元，并按并行分布式方式工作，具有较强的演绎、推理、联想、学习功能和形象思维能力。例如，对图像、图形、景物等，人类可凭直觉、视觉，通过视网膜、脑神经对其进行快速响应与处理，而传统计算机却显得非常迟钝。

如何缩小这种差距呢？要靠人工智能技术。从长远观点看，需要彻底改变冯·诺依曼计算机的体系结构，研制智能计算机。但从目前条件看，还主要靠智能程序系统来提高现有计算机的智能化程度。智能程序系统和传统的程序系统相比，具有以下几个主要特点。

1. 基于知识

知识是一切智能系统的基础，任何智能系统的活动过程都是一个获取知识和运用知识的过程，而要获取和运用知识，首先应该能够对知识进行表示。所谓知识表示就是用某种约定的方式对知识进行的描述。在知识表示方面目前有两种基本观点：一种是叙述性（declarative）观点，另一种是过程性（procedural）观点。叙述性知识表示观点是将知识的表示与知识的运用分开处理，在知识表示时不涉及如何运用知识的问题；过程性知识表示观点是将知识的表示与知识的运用结合起来，知识就包含在程序之中。两种观点各有利弊，目前人工智能程序采用较多的是叙述性观点。当然，也可根据具体问题的性质而定。

2. 运用推理

所谓推理（reasoning）就是根据已有知识运用某种策略推出新知识的过程。事实上，一个智能系统仅有知识是不够的，它还必须具有思维能力，即能够运用知识进行推理和解决问题。人工智能中的推理方法主要有经典逻辑推理、不确定性推理和非单调性推理。

3. 启发式搜索

所谓搜索就是根据问题的现状不断寻找可利用的知识，使问题能够得以解决的过程。人工智能中的搜索分为盲目搜索和启发式（heuristics）搜索两种。所谓盲目搜索是指仅按预定策略进行搜索，搜索中获得的信息不改变搜索过程的搜索方法。所谓启发式搜索则是指能够利用搜索中获得的问题本身的一些特性信息（也称启发信息）来指导搜索过程，使搜索朝着最有希望的方向前进。人工智能主要采用的是启发式搜索策略。

4. 数据驱动方式

所谓数据驱动（data driven）是指在系统处理的每一步，当考虑下一步该做什么时，需要根据此前所掌握的数据内容（也称事实）来决定。与数据驱动方式对应的另一种方式是程序驱动（program driven），所谓程序驱动是指系统处理的每一步及下一步该做什么都是由程序事先预定好的。人类在解决问题时主要使用数据驱动方式，因此智能程序系统也应该使用数据驱动方式，这样会更接近于人类分析问题、解决问题的习惯。

5. 用人工智能语言建造系统

人工智能语言是一类适应于人工智能和知识工程领域的、具有符号处理和逻辑推理能力的计算机程序语言。它能够完成非数值计算、知识处理、推理、规划、决策等具有智能的各种复杂问题的求解。人工智能语言 and 传统程序设计语言相比，具有以下主要特点。

- （1）具有回溯和非确定性推理功能；
- （2）能够进行符号形式的知识信息处理；
- （3）能够动态使用知识和动态分配存储空间；
- （4）具有模式匹配和模式调用功能；
- （5）具有并行处理和并行分布式处理功能；
- （6）具有信息隐蔽、抽象数据类型、继承、代码共享及软件重用等面向对象的特征；
- （7）具有解释推理过程的说明功能；
- （8）具有自学习、自适应的开放式软件环境等。

人工智能语言可从总体上划分为通用型和专用型两种。通用型人工智能语言主要是指以 LISP 为代表的函数型语言、以 Prolog 为代表的逻辑性语言和以 C++ 等为代表的面向对象语言。专用型人工智能语言主要是指那些由多种人工智能语言或过程语言相互结合而构成的，具有解决多种问题能力的专家系统开发工具和人工智能开发环境。

1.4 人工智能的研究领域

在大多数学科中都存在着几个不同的研究领域，每个领域都有其特有的感兴趣的研究课题、研究技术和术语。由于智能的复杂性，人工智能实际上是一个大学科。经过 40 余年的发展，现在其技术脉络已日趋清楚，理论体系已逐渐形成，应用范围不断扩展。人工智能学科现已分化出了许多的分支研究领域。下面我们从不同角度对其进行简介。

1.4.1 经典的人工智能研究领域

在经典的人工智能研究中，这样的领域包括逻辑推理与定理证明、博弈、自然语言处理、专家系统、自动程序设计、机器学习、人工神经网络、机器人学、模式识别、计算机视觉、智能控制、智能检索、智能调度与指挥、智能决策支持系统、知识发现和数据挖掘，以及分布式人工智能等。

值得指出的是，正如不同的人工智能子领域不是完全独立的一样，这里所要讨论的各种智能特性也完全不是互不相关的。把它们分开来介绍只是为了便于指出现有的人工智能程序能够做些什么和还不能做什么。大多数人工智能研究课题都涉及许多（如果不是全部的话）智能领域。

1. 逻辑推理与定理证明

早期的逻辑演绎研究工作与问题和难题的求解相当密切。已经开发出的程序能够借助于对事实数据库的操作来“证明”断定，其中每个事实由分立的数据结构表示，就像数学逻辑中由分立公式表示一样。与人工智能的其他技术的不同之处是，这些方法能够完整地、一致地加以表示。也就是说，只要本原事实是正确的，那么程序就能够证明这些从事实得出的定理，而且也仅仅是证明这些定理。

逻辑推理是人工智能研究中最持久的子领域之一。其中特别重要的是要找到一些方法，只把注意力集中在一个大型数据库中的有关事实上，留意可信的证明，并在出现新信息时适时修正这些证明。

为数学中臆测的定理寻找一个证明或反证，确实称得上是一项智能任务。为此不仅需要有能力根据假设进行演绎的能力，而且需要某些直觉技巧。例如，为了求证主要定理而猜测应当首先证明哪个引理。一个熟练的数学家运用他的（以大量专门知识为基础的）判断力能够精确地推测出某个科目范围里哪些前已证明的定理在当前的证明中是有用的，并把他的主问题分解为若干子问题，以便独立地处理它们。有几个定理证明程序已在有限的程度上具有某些这样的技巧。1976 年 7 月，美国的阿佩尔（K. Appel）等人合作解决了长达 124 年之久的难题——四色定理。他们用 3 台大型计算机，花去 1200h 的 CPU 时间，并对中间结果进行人为反复修改 500 多处。四色定理的成功证明曾轰动计算机界。

定理证明的研究在人工智能方法的发展中曾经产生过重要的影响。例如，采用谓词逻辑语言的演绎过程的形式化有助于更清楚地理解推理的某些子命题。许多非形式化的工作，

包括医疗诊断和信息检索都可以和定理证明问题一样加以形式化。因此，在人工智能方法的研究中定理证明是一个极其重要的论题。

2. 博弈

博弈 (game playing) 是一个有关对策和斗智问题的研究领域。例如，下棋、打牌、战争等这一类竞争性智能活动都属于博弈问题。博弈是人类社会和自然界中普遍存在的一种现象，博弈的双方可以是个人、群体，也可以是生物群或智能机器，各方都力图用自己的智力击败对方。

人工智能的第1个大成就是发展了能够求解难题的下棋（如国际象棋）程序。在下棋程序中应用的某些技术，如向前看几步，并把困难的问题分成一些比较容易的子问题，发展成为搜索和问题归约这样的人工智能基本技术。今天的计算机程序能够下锦标赛水平的各种方盘棋、十五子棋和国际象棋。另一种问题求解程序把各种数学公式符号汇编在一起，其性能达到很高的水平，并正在为许多科学家和工程师所应用。有些程序甚至还能够用经验来改善其性能。1993年，美国出版了一个叫做 MACSYMA 的软件，就能进行比较复杂的数学公式符号运算。

这个领域中未解决的问题包括人类棋手具有的、但尚不能明确表达的能力，如国际象棋大师们洞察棋局的能力。另一个未解决的问题涉及问题的原概念，在人工智能中叫做问题表示的选择。人们常常能够找到某种思考问题的方法从而使求解变易而解决该问题。到目前为止，人工智能程序已经知道如何考虑它们要解决的问题，即搜索解答空间，寻找较优的解答。

迄今为止，人工智能对博弈的研究多以下棋为对象，但其目的并不是为了让计算机与人下棋，而主要是为了给人工智能研究提供一个试验场地，对人工智能的有关技术进行检验，从而也促进这些技术的发展。博弈研究的一个代表性成果是 IBM 公司研制的超级计算机“深蓝”。“深蓝”被称为世界上第1台超级国际象棋计算机，该机有32个独立运算器，其中每个运算器的运算速度都在每秒200万次以上，机内还装了一个包含有200万个棋局的国际象棋程序。“深蓝”于1997年5月3日至5月11日，在美国纽约曼哈顿同当时的国际象棋世界冠军苏联人卡斯帕罗夫对弈6局，结果“深蓝”获胜。

3. 自然语言理解

自然语言处理也是人工智能的早期研究领域之一，并引起进一步的重视。已经编写出能够从内部数据库回答用英语提出的问题的程序，这些程序通过阅读文本材料和建立内部数据库，能够把句子从一种语言翻译为另一种语言，执行用英语给出的指令和获取知识等。有些程序甚至能够在一定程度上翻译从话筒输入的口头指令（而不是从键盘键入计算机的指令）。尽管这些语言系统并不像人们在语言行为中所做的那样好，但是它们能够适合某些应用。那些能够回答一些简单询问的和遵循一些简单指示的程序是这方面的初期成就，它们与机器翻译初期出现的故障一起，促使整个人工智能语言方法的彻底变革。目前语言处理研究的主要课题是：在翻译句子时，以主题和对话情况为基础，注意大量的一般常识——世界知识和期望作用的重要性。

实际语言系统的技术发展水平是用各种软件系统的有效“前端”来表示的。这些程序

接收某些局部形式的输入，但不能处理英语语法的某些微小差别，而且只适用于翻译某个有限讲话范围内的句子。人工智能在语言翻译与语音理解程序方面已经取得的成就，发展为人类自然语言处理的新概念。

当人们用语言互通信息时，他们几乎不费力地进行极其复杂却又只需要一点点理解的过程。然而要建立一个能够生成和“理解”哪怕是片断自然语言的计算机系统却是异常困难的。语言已经发展成为智能动物之间的一种通信媒介，它在某些环境条件下把一点“思维结构”从一个头脑传输到另一个头脑，而每个头脑都拥有庞大的高度相似的周围思维结构作为公共的文本。这些相似的、前后有关的思维结构中的一部分允许每个参与者知道对方也拥有这种共同结构，并能够在通信“动作”中用它来执行某些处理。语言的发展显然为参与者使用他们巨大的计算资源和公共知识来生成和理解高度压缩和流畅的知识开拓了机会。语言的生成和理解是一个极为复杂的编码和解码问题。

一个能理解自然语言信息的计算机系统看起来就像一个人一样需要有上下文知识以及根据这些上下文知识和信息用信息发生器进行推理的过程。理解口头的和书写的片断语言的计算机系统所取得的某些进展，其基础就是有关表示上下文知识结构的某些人工智能思想以及根据这些知识进行推理的某些技术。

4. 专家系统

专家系统（expert system, ES）是一种基于知识的智能系统，它将领域专家的经验用知识表示方法表示出来，并放入知识库中，供推理机使用。由于专家系统包含了大量的专家知识，并具有使用这些知识的能力，因此可用来解决该领域中需要专家才能解决的问题。专家系统目前尚无公认的定义，一种比较一致的解释是：专家系统是一个能在某特定领域内，以专家水平去解决该领域中困难问题的计算机程序。

一般地说，专家系统是一个智能计算机程序系统，其内部具有大量专家水平的某个领域的知识与经验，能够利用人类专家的知识 and 解决问题的方法来解决该领域的问题。也就是说，专家系统是一个具有大量专门知识与经验的程序系统，它应用人工智能技术，根据某个领域一个或多个人类专家提供的知识和经验进行推理和判断，模拟人类专家的决策过程，以解决那些需要专家决定的复杂问题。

近年来，在专家系统或“知识工程”的研究中已经出现了成功和有效地应用人工智能技术的趋势。有代表性的是，用户与专家系统进行“咨询对话”，就像他与具有某方面经验的专家进行对话一样：解释他的问题，建议进行某些试验以及向专家系统提出询问以求得到有关解答等。目前的实验系统在咨询任务，如化学和地质数据分析、计算机系统结构、建筑工程以及医疗诊断等方面，其质量已经达到很高的水平。可以把专家系统看作人类专家（他们用“知识获取模型”与专家系统进行人一机对话）和人类用户（他们用“咨询模型”与专家系统进行人一机对话）之间的媒介。在人工智能的这个领域里，还有许多研究集中在使专家系统具有解释它们的推理能力，从而使咨询更好地为用户所接受，又能帮助人类专家发现系统推理过程中出现的差错。

当前的研究涉及有关专家系统设计各种问题。这些系统是在某个领域的专家（他可能无法明确表达他的全部知识）与系统设计者之间经过艰苦的反复交换意见之后建立起来的。现有的专家系统都局限在一定范围内，而且没有人类那种能够知道自己什么时候可能

出错的感觉，新的研究包括应用专家系统来教初学者以及请教有经验的专业人员。

自动咨询系统向用户提供特定学科领域内的专家结论。在已经建立的专家咨询系统中，有能够诊断疾病的（包括中医诊断智能机）、估计潜在石油等矿藏的、研究复杂有机化合物结构的以及提供使用其他计算机系统的参考意见等。

发展专家系统的关键是表达和运用专家知识，即来自人类专家的并已被证明对解决有关领域内的典型问题是有用的事实和过程。专家系统和传统的计算机程序最本质的不同之处在于专家系统所要解决的问题一般没有算法解，并且经常要在不完全、不精确或不确定的信息基础上做出结论。

专家系统可以解决的问题一般包括解释、预测、诊断、设计、规划、监视、修理、指导和控制等。高性能的专家系统也已经从学术研究开始进入实际应用研究。专家系统作为人工智能中最活跃、发展最快的一个分支，已广泛应用于工业、农业、医学、地质、气象、交通、军事、法律、空间技术、环境科学和信息管理等众多领域，并产生了巨大的经济效益和社会效益。

随着人工智能整体水平的提高，专家系统也获得发展。正在开发的新一代专家系统有分布式专家系统和协同式专家系统等。在新一代专家系统中，不但采用基于规则的方法，而且采用基于模型的原理。

专家系统自出现以来已经历了几个发展阶段，目前正在向多专家协同的分布式专家系统方向发展。

5. 自动程序设计

自动程序设计也许并不是人类知识的一个十分重要的方面，但是它本身却是人工智能的一个重要研究领域。已经研制出能够以各种不同的目的描述（例如，输入输出对，高级语言描述，甚至英语描述算法）来编写计算机程序。这方面的进展局限于少数几个完全现成的例子。对自动程序设计的研究不仅可以促进半自动软件开发系统的发展，而且也使通过修正自身数码进行学习（即修正它们的性能）的人工智能系统得到发展。程序理论方面的有关研究对人工智能的所有研究工作都是很重要的。

编写一段计算机程序的任务既同定理证明又同机器人学有关。自动程序设计、定理证明和机器人问题求解中大多数基础研究是相互重叠的。在某种意义上讲，编译程序已经在干“自动程序设计”的工作。编译程序接受一份有关想干些什么的完整的源码说明，然后编写一份目标码程序去实现。这里所指的自动程序设计是某种“超级编译程序”，或者是某种能够对程序要实现什么目标进行非常高级描述的程序，并能够由这个程序产生出所需要的新程序。这种高级描述可能是采用形式语言的一条精辟语句（如谓词演算），也可能是一种松散的描述（如用英语），这就要求在系统和用户之间进一步对话澄清语言的模糊。

自动编制一份程序来获得某种指定结果的任务同证明一份给定程序将获得某种指定结果的任务是紧密相关的。后者叫做程序验证。许多自动程序设计系统将产生一份输出程序的验证作为额外收获。

自动程序设计研究的重大贡献之一是作为问题求解策略的调整概念。已经发现，对程序设计或机器人控制问题，先产生一个不费事的有错误的解，然后再修改它（使它正确工作），这种做法一般要比坚持要求第1个解就完全没有缺陷的做法有效得多。

6. 机器学习

学习能力无疑是人工智能研究上最突出和最重要的一个方面。人工智能在这方面的研究近年来取得了一些进展。

学习是人类智能的主要标志和获得知识的基本手段。机器学习（自动获取新的事实及新的推理算法）是使计算机具有智能的根本途径。正如香克（R. Shank）所说：“一台计算机若不会学习，就不能称为具有智能的。”此外，机器学习还有助于发现人类学习的机理和揭示人脑的奥秘。所以这是一个始终得到重视，理论正在创立，方法日臻完善，但远未达到理想境地的研究领域。

学习是一个有特定目的的知识获取过程，其内部表现为新知识结构的不断建立和修改，而外部表现为性能的改善。传统的机器学习倾向于使用符号表示而不是数值表示，使用启发式方法而不是算法。传统机器学习的另一倾向是使用归纳（induction）而不是演绎（deduction）。前一倾向使它有别于人工智能的模式识别等分支；后一倾向使它有别于定理证明等分支。

一个学习过程本质上是学习系统把导师（或专家）提供的信息转换成能被系统理解并应用的形式。按系统对导师的依赖程度可将学习方法分为如下几种。

（1）机械式学习（rote learning）。

导师要完成全部转换工作，系统只负责存储。

（2）讲授式学习（learning from instruction）。

学习系统对导师提供的信息有一定的选择能力，并予以形式化。目前大多数基于知识的系统用这种方法建立知识库。

（3）类比学习（learning by analogy）。

已知源域与目标域这样两个不同的领域，且知道这两个域中已有某些满足相似度量度的知识与求解方法，则通过类比学习能将源域中的知识与求解方法转换到目标域中去。例如，著名的卢瑟福类比就是通过将原子结构（目标）同太阳系（源）进行类比，从而发现原子结构的奥秘。

（4）归纳学习（learning from induction）。

环境所提供的是关于大量实例的输入和输出描述，学习元进行推理归类和对共性的分析，抽象出一般的概念和规则，使这些新概念、新规则能蕴含所有实例。这种学习所接受的实例中，不仅有正例，还可能有反例，但这些反例是已被告知是错误的，故它们不属于噪音或矛盾，它们对学习的作用，甚至可能比正例还重要。

（5）观察发现式学习（learning by observation & discovery）。

它是归纳学习的高一层次，它所接受的实例中含有噪音和矛盾，需由学习元对它们鉴别、提纯，而且对实例间的相互联系进行分析，实现概念聚类，或发现新的概念和定律，因而有创新的成分。

用第（3），（4）和（5）这3种学习方法开发学习系统，目前还处于探索阶段，它们一般是局限在特定领域中为探索学习机制而开发的。目前运用较多的还是前两种，这也呼应了前面所指出的“远未达到理想境地”的评语。

此外，近年来又发展了下列各种学习方法：

- (6) 基于解释的学习。
- (7) 基于事例的学习。
- (8) 基于概念的学习。
- (9) 基于神经网络的学习。
- (10) 遗传学习等。

7. 人工神经网络

由于冯·诺依曼（Van Neumann）体系结构的局限性，数字计算机存在一些尚无法解决的问题。例如，基于逻辑思维的知识处理，在一些比较简单的知识范畴内能够建立比较清楚的理论框架，部分地表现出人的某些智能行为，但是，在视觉理解、直觉思维、常识与顿悟等问题上显得力不从心。这种做法与人类智能活动有许多重要差别。传统的计算机不具备学习能力，无法快速处理非数值计算的形象思维等问题，也无法求解那些信息不完整、不确定性和模糊性的问题。人们一直在寻找新的信息处理机制，神经网络计算就是其中之一。

研究结果已经证明，用神经网络处理直觉和形象思维信息具有比传统处理方式好得多的效果。神经网络的发展有着非常广阔的科学背景，是众多学科研究的综合成果。神经生理学家、心理学家与计算机科学家共同研究得出的结论是：人脑是一个功能特别强大、结构异常复杂的信息处理系统，其基础是神经元及其互联关系。因此，对人脑神经元和人工神经网络的研究，可能创造出新一代人工智能机——神经计算机。

对神经网络的研究始于20世纪40年代初期，经历了一条十分曲折的道路，几起几落，20世纪80年代初以来，对神经网络的研究再次出现高潮。霍普菲尔德（Hopfield）提出用硬件实现神经网络，鲁梅尔哈特（Rumelhart）等提出多层网络中的反向传播（BP）算法就是两个重要标志。

对神经网络模型、算法、理论分析和硬件实现的大量研究，为神经网络计算机走向应用提供了物质基础。现在，神经网络已在模式识别、图像处理、组合优化、自动控制、信息处理、机器人学和人工智能的其他领域获得日益广泛的应用。人们期望神经计算机将重建人脑的形象，极大地提高信息处理能力，在更多方面取代传统的计算机。

8. 机器人学

人工智能研究日益受到重视的另一个分支是机器人学，其中包括对操作机器人装置程序的研究。这个领域所研究的问题，从机器人手臂的最佳移动到实现机器人目标的动作序列的规划方法，无所不包。尽管已经建立了一些比较复杂的机器人系统，不过目前正在工业上运行的成千上万台机器人，都是一些按预先编好的程序执行某些重复作业的简单装置。大多数工业机器人是“盲人”，只有某些机器人能够用电视摄像机来“看”。电视摄像机发送一组信息返回计算机。处理视觉信息是人工智能另一个十分活跃和十分困难的研究领域。已经开发的程序能够识别可见景物的实体与阴影，甚至能够辨别出两幅图像间（例如，在航空侦察中）的细小差别。

一些并不复杂的动作控制问题，如移动式机器人的机械动作控制问题，表面上看并不需要很多智能。即使是个小孩，也能顺利地通过周围环境，操作电灯开关、玩具积木和餐

具等物品。然而人类几乎下意识就能完成的这些任务，要是由机器人来实现就要求机器人具备在求解需要较多智能的问题时所具有的能力。

机器人和机器人学的研究促进了许多人工智能思想的发展。它所导致的一些技术可用来模拟世界的状态，用来描述从一种世界状态转变为另一种世界状态的过程。它对于怎样产生动作序列的规划以及怎样监督这些规划的执行有了一种较好的理解。复杂的机器人控制问题迫使人们发展一些方法，先在抽象和忽略细节的高层进行规划，然后再逐步在细节越来越重要的低层进行规划。

智能机器人的研究和应用体现出广泛的学科交叉，涉及众多的课题，如机器人体系结构、机构、控制、智能、视觉、触觉、力觉、听觉、机器人装配、恶劣环境下的机器人以及机器人语言等。机器人已在工业、农业、商业、旅游业、空中和海洋以及国防等领域获得越来越普遍的应用。

9. 模式识别

计算机硬件的迅速发展，计算机应用领域的不断开拓，急切地要求计算机能更有效地感知诸如声音、文字、图像、温度、震动等人类赖以发展自身、改造环境所运用的信息资料。但就一般意义来说，目前计算机却无法直接感知它们，键盘、鼠标等外部设备，对于这样五花八门的外部世界显得无能为力。纵然电视摄像机、图文扫描仪、话筒等硬设备业已解决了上述非电信号的转换，并与计算机联机，但由于识别技术不高，而未能使计算机真正知道所采录的究竟是什么信息。计算机对外部世界感知能力的低下，成为开拓计算机应用的狭窄瓶颈，也与其高超的运算能力形成强烈的对比。于是，着眼于拓宽计算机的应用领域，提高其感知外部信息能力的学科——模式识别便得到迅速发展。

“模式（pattern）”一词的本意是指完美无缺的供模仿的一些标本。于是，模式识别就是指识别出给定物体所模仿的标本。人们生产和生活都离不开模式识别。但人工智能所研究的模式识别是指用计算机代替人类或帮助人类感知模式，是对人类感知外界功能的模拟，研究的是计算机模式识别系统，也就是使一个计算机系统具有模拟人类通过感官接受外界信息、识别和理解周围环境的感知能力。

实验表明，人类接受外界信息的80%以上来自视觉，10%左右来自听觉。所以，早期的模式识别研究工作集中在对文字和二维图像的识别方面，并取得了不少成果。自20世纪60年代中期起，机器视觉方面的研究工作开始转向解释和描述复杂的三维景物这一更困难的课题上。罗伯斯特（Robest）于1965年发表的论文，指出了分析由棱柱体组成的物景的方向，迈出了用计算机把三维图像解释成三维物景的一个单眼视图的第1步，即所谓的积木世界。

接着机器识别由积木世界进入识别更复杂的物景和在复杂环境中寻找目标以及室外物景分析等方面的研究。目前研究的热点是活动目标的识别和分析，它是景物分析走向实用化研究的一个标志。

语音识别技术的研究始于20世纪50年代初期。1952年，美国贝尔实验室的戴维斯（Davis）等人成功地进行了0~90个数字的语音识别实验，其后由于当时技术上的困难，研究进展缓慢，直到1962年才由日本研制成功第1个连续多位数字语音识别装置。1969年，日本的板仓斋藤提出了线性预测方法，对语音识别和合成技术的发展起到了推动作用。

20 世纪 70 年代以来, 各种语音识别装置相继出现, 性能良好的能识别单词的声音识别系统已进入实用阶段。神经网络用于语音识别也已取得成功。

模式识别是一门不断发展的新学科, 它的理论基础和研究范围也在不断发展。随着生物医学对人类大脑的初步认识, 模拟人脑构造的计算机实验, 即人工神经网络方法早在 20 世纪 50 年代末、60 年代初就已经开始。至今, 在模式识别领域, 神经网络方法已经成功地用于手写字符的识别、汽车牌照的识别、指纹识别、语音识别等方面。目前模式识别学科正处于大发展的阶段。随着应用范围的不断扩大, 随着计算机科学的不断进步, 基于人工神经网络的模式识别技术, 在本世纪将有更大的发展。

10. 计算机视觉

计算机视觉或机器视觉已从模式识别的一个研究领域发展为一门独立的学科。在视觉方面, 已经给计算机系统装上电视输入装置以便能够“看见”周围的东西。视觉是感知问题之一。在人工智能中研究的感知过程通常包含一组操作。例如, 可见的景物由传感器编码, 并被表示为一个灰度数值的矩阵。这些灰度数值由检测器加以处理。检测器搜索主要图像的成分, 如线段、简单曲线和角度等。这些成分又被处理, 以便根据景物的表面和形状来推断有关景物的三维特性信息。其最终目标则是利用某个适当的模型来表示该景物。

整个感知问题的要点是形成一个精练的表示以取代难以处理的、极其庞大的未经加工的输入数据。最终表示的性质和质量取决于感知系统的目标。不同系统有不同的目标, 但所有系统都必须把来自输入的多得惊人的感知数据简化为一种易于处理的和有意义的描述。

对不同层次的描述做出假设, 然后测试这些假设。这一策略为视觉问题提供了一种方法。已经建立的某些系统能够处理一幅景物的某些适当部分, 以此扩展一种描述若干成分的假设。然后这些假设通过特定的场景描述检测器进行测试。这些测试的结果又用来发展更好的假设。

计算机视觉通常可分为低层视觉与高层视觉两类。并非人工智能的全部领域都是围绕着知识处理的, 计算机低层视觉就是一例。低层视觉主要执行预处理功能, 如边缘检测、动目标检测、纹理分析, 通过阴影获得形状、立体造型、曲面色彩等。其目的是使被观察的对象更突显出来, 这时还谈不到对它的理解。高层视觉则主要是理解所观察的形象, 也只有这时才显示出掌握与所观察的对象相关联的知识的重要性。

机器视觉的前沿研究领域包括实时并行处理、主动式定性视觉、动态和时变视觉、三维景物的建模与识别、实时图像压缩传输和复原、多光谱和彩色图像的处理与解释等。机器视觉已在机器人装配、卫星图像处理、工业过程监控、飞行器跟踪和制导以及电视实况转播等领域获得极为广泛的应用。

11. 智能控制

人工智能的发展促进自动控制向智能控制发展。智能控制是一类无须（或需要尽可能少的）人的干预就能够独立地驱动智能机器实现其目标的自动控制。或者说, 智能控制是驱动智能机器自主地实现其目标的过程。许多复杂的系统, 难以建立有效的数学模型和用

常规控制理论进行定量计算与分析,而必须采用定量数学解析法与基于知识的定性方法的混合控制方式。随着人工智能和计算机技术的发展,已可能把自动控制和人工智能以及系统科学的某些分支结合起来,建立一种适用于复杂系统的控制理论和技术。智能控制正是在这种条件下产生的。它是自动控制的最新发展阶段,也是用计算机模拟人类智能的一个重要研究领域。

1965年,傅京孙首先提出把人工智能的启发式推理规则用于学习控制系统。十多年后,建立实用智能控制系统的技术逐渐成熟。1971年,傅京孙提出把人工智能与自动控制结合起来的的思想。1977年,美国G. N. 萨里迪斯提出把人工智能、控制论和运筹学结合起来的的思想。1986年,中国的蔡自兴提出把人工智能、控制论、信息论和运筹学结合起来的的思想。按照这些结构理论已经研究出一些智能控制的理论和技术,用来构造用于不同领域的智能控制系统。1985年,在美国首次召开了智能控制学术讨论会。1987年,在美国召开了首届智能控制国际学术会议,它标志着智能控制作为一个新的学科分支得到承认。1993年,在北京召开的第一届全球华人智能控制与智能自动化大会是国际智能控制界的又一盛事,并推动了国内外智能控制与智能自动化研究的发展。

智能控制是同时具有以知识表示的非数学广义世界模型和数学公式模型表示的混合控制过程,也往往是含有复杂性、不完全性、模糊性或不确定性以及不存在已知算法的非数学过程,并以知识进行推理,以启发来引导求解过程。因此,在研究和设计智能控制系统时,不把注意力放在数学公式的表达、计算和处理方面,而是放在对任务和世界模型的描述、符号和环境的识别以及知识库和推理机的设计开发上,即放在智能机模型上。智能控制的核心在高层控制,即组织级控制。其任务在于对实际环境或过程进行组织,即决策和规划,以实现广义问题求解。为此,需要采用符号信息处理、启发式程序设计、知识表示以及自动推理和决策等相关技术。这些问题求解过程与人脑的思维过程具有一定的相似性,即具有一定程度的“智能”。已经提出的用以构造智能控制系统的理论和技术有分级递阶控制理论、分级控制器设计的熵方法、智能逐级增高而精度逐级降低原理、专家控制系统、学习控制系统和基于NN的控制系统等。分级递阶智能控制系统和专家控制系统是两种最重要的智能控制系统。

智能控制有很多研究领域,它们的研究课题既具有独立性,又相互关联。目前研究得较多的是以下6个方面:智能机器人规划与控制、智能过程规划、智能过程控制、专家控制系统、语音控制以及智能仪器。

智能控制是一门形成不久的新生学科,无论在理论上或实践上,都还很成熟、很不完善,有待进一步研究与发展。作为当今自动控制最高水平的智能控制,近年来已获迅速发展,应用日益普遍,并已引起高度重视。尽管在智能控制方面的每一进展都可能要付出艰苦劳动和昂贵代价,然而随着人工智能技术、机器人技术、航天技术、海洋工程、计算机集成制造技术和计算机技术的迅速发展,智能控制必将迎来它的发展新时期,为自动化科学技术的发展谱写新篇章。

12. 智能检索

随着科学技术的迅速发展,出现了“信息爆炸”的情况。对国内外种类繁多和数量巨大的科技文献之检索远非人力和传统检索系统所能胜任。研究智能检索系统已成为科技持

续快速发展的重要条件。

数据库系统是存储某学科大量事实的计算机软件系统，它们可以回答用户提出的有关该学科的各种问题。例如，假设这些事实是某公司的人事档案，这个数据库中的某些条款可以代表下列事实：“张强在采购部工作”，“张强在1995年8月15日退休”，“采购部共有15名工作人员”和“李明是采购部经理”等。

数据库系统的设计也是计算机科学的一个活跃的分支。为了有效地表示、存储和检索大量事实，已经发展了许多技术。当想用数据库中的事实进行推理并从中检索答案时，这个课题就显得很有意义。

智能信息检索系统的设计者们将面临以下几个问题。首先，建立一个能够理解以自然语言陈述的询问系统本身就存在不少问题。其次，即使能够通过规定某些机器能够理解的形式化询问语句来回避语言理解问题，但仍然存在一个如何根据存储的事实演绎出答案的问题。第三，理解询问和演绎答案所需要的知识都可能超出该学科领域数据库所表示的知识。常识往往是需要的，但在学科领域的数据库中常常被忽略掉。例如，在前述的人事档案中，一个智能系统应能对询问“谁是张强的领导？”演绎出答案“李明”。为此，这个系统就必须知道一个部门的经理就是该部门工作人员的领导这一常识。怎样表示和应用常识是采用人工智能方法的系统设计问题之一。

13. 智能调度与指挥

确定最佳调度或组合的问题是人们感兴趣的又一类问题。一个经典的问题就是推销员旅行问题。这个问题要求为推销员寻找一条最短的旅行路线。他从某个城市出发，访问每个城市一次，且只许一次，然后回到出发的城市。这个问题的一般提法是：对由 n 个结点组成的一个图的各条边，寻找一条最小费用的路径，使得这条路径对 n 个结点的每个点只许穿过一次。

许多问题具有这类相同的特性。八皇后问题就是其中之一。这个问题要求在一个标准的国际象棋棋盘上按下列要求放置8个皇后，没有一个皇后可以捕获任何其他皇后，即在任何一行、一列或一对角线上最多只能放置一个皇后。大多数这类问题能够从可能的组合或序列中选取一个答案，不过组合或序列的范围很大。试图求解这类问题的程序产生了一种组合爆炸的可能性。这时，即使是大型计算机的容量也会被用光。

在这些问题中有几个（包括推销员旅行问题）是属于计算理论家称为NP完全一类的问题。他们根据理论上的最佳方法计算出所耗时间（或所走步数）的最坏情况来排列不同问题的难度。该时间或步数是随着问题大小的某种量度（在推销员旅行问题中，城市数目就是问题大小的一种量度）增长的。譬如说，问题的难度将随着问题大小按线性，或多项式，或指数方式增长。

人工智能学家们曾经研究过若干组合问题的求解方法。他们的努力集中在使“时间—问题大小”曲线的变化尽可能缓慢地增长，即使是必须按指数方式增长。有关问题域的知识再次成为比较有效的求解方法的关键。为处理组合问题而发展起来的许多方法对其他组合上不甚严重的问题也是有用的。

智能组合调度与指挥方法已被应用于汽车运输调度、列车的编组与指挥、空中交通管制以及军事指挥等系统。它已引起有关部门的重视。

14. 智能决策支持系统

智能决策支持系统(intelligent decision support systems, IDSS)是决策支持系统(decision support systems, DSS)与人工智能相结合的产物,它将人工智能中的知识表示与处理的思想引入到DSS,其独特的研究方法和广泛的发展前途使之一出现就成为决策支持技术研究的热点。

智能决策支持系统是以信息技术为手段,应用管理科学、计算机科学及有关学科的理论和方法,针对半结构化和非结构化的决策问题,通过提供背景材料、协助明确问题、修改完善模型、列举可能方案、进行分析比较等方式,为管理者做出正确决策提供帮助的智能型人机交互信息系统。

实践表明,只有当决策支持系统具有较丰富的知识和较强的知识处理能力时,才能向决策者提供更为有效的决策支持。

15. 知识发现和数据挖掘

知识发现(knowledge discovery in database)和数据挖掘(data mining)是在数据库的基础上实现的一种知识发现系统。它是通过综合运用统计学、粗糙集、模糊数学、机器学习和专家系统等多种学习手段和方法,从数据库中提炼和抽取知识,从而揭示出蕴含在这些数据背后的客观世界的内在联系和本质原理,实现知识的自动获取。

传统的数据库技术仅限于对数据库的查询和检索,不能从数据库中提取知识,使得数据库中所蕴含的丰富知识被白白浪费。知识发现和数据挖掘以数据库作为知识源去抽取知识,不仅可以提高数据库中数据的利用价值,同时也为专家系统的知识获取开辟了一条新的途径。

16. 分布式人工智能

分布式人工智能(distributed artificial intelligence, DAI)是随着计算机网络、计算机通信和并发程序设计技术而发展起来的一个新的人工智能研究领域。它主要研究在逻辑或物理上分散的智能系统之间如何相互协调各自的智能行为,实现问题的并行求解。分布式人工智能的研究为在计算机网络环境下设计和建立大型复杂智能系统提供了一条有效途径,体现了新一代软件设计的思想,是当前人工智能研究的一个热点。

分布式人工智能的研究目前有两个主要方向:一个是分布式问题求解,另一个是多智能主体系统。分布式问题求解的主要任务是要创建一个可以对某一问题进行共同求解的协作群体;多智能主体系统不限于单一目标,其主要任务是要创建一个多智能主体之间能够相互协调智能行为的、可以共同处理单个目标和多个目标的智能群体。

1.4.2 基于脑功能模拟的领域划分

按照人脑的功能模拟,可以将人工智能的研究领域划分为机器感知、机器联想、机器推理、机器学习、机器理解、机器行为等。

1. 机器感知

机器感知就是计算机直接“感觉”周围世界。具体来讲，就是计算机像人一样通过“感觉器官”直接从外界获取信息。如通过视觉器官获取图形、图像信息，通过听觉器官获取声音信息。所以，要使机器具有感知能力，就首先必须给机器配置各种感觉器官，如视觉器官、听觉器官、嗅觉器官等。于是，机器感知还可以再分为机器视觉、机器听觉等。

要研究机器感知，首先要涉及图像、声音等信息的识别问题。为此，现在已发展了一门称为“模式识别”的专门学科，模式识别的主要目标就是用计算机来模拟人的各种识别能力，当前主要是对视觉能力和听觉能力的模拟，并且主要集中于图形识别和语音识别。

图形识别主要是研究各种图形（如文字、符号、图形、图像和照片等）的分类。例如，识别各种印刷体和某些手写体文字，识别指纹、白血球和癌细胞等。这方面的技术已经进入实用阶段。

语音识别主要是研究各种语音信号的分类。语音识别技术近年来发展很快，现已有商品化产品（如汉字语音录入系统）上市。

模式识别的过程大体是先将摄像机、送话器或其他传感器接受的外界信息转变成电信号序列，计算机再进一步对这个电信号序列进行各种预处理，从中抽出有意义的特征，得到输入信号的模式，然后与机器中原有的各个标准模式进行比较，完成对输入信息的分类识别工作。

机器感知不仅是对人类感知的模拟，也是对人类感知的延伸。因为人的感知能力是很有限的，例如，对声音的感知只能限于一定的声波频率范围。在这一点，人的感觉灵敏度还不如有些高等动物甚至昆虫。那么可想而知，若计算机的感知能力一旦实现，则必将超过人类自身。

2. 机器联想

仔细分析人脑的思维过程可以发现，联想实际是思维过程中最基本、使用最频繁的一种功能。例如，当听到一段乐曲，头脑中可能会立即浮现出几十年前的某个场景，甚至一段往事，这就是联想。所以，计算机要模拟人脑的思维就必须具有联想功能。要实现联想无非就是建立事物之间的联系，在机器世界里面就是有关数据、信息或知识之间的联系。当然，建立这种联系的办法很多，比如用指针、函数、链表等。通常的信息查询就是这样做的。但传统方法实现的联想，只能对于那些完整的、确定的（输入）信息，联想起（输出）有关的信息。这种“联想”与人脑的联想功能相差甚远。人脑能对那些残缺的、失真的、变形的输入信息，仍然快速准确地输出联想响应。例如，对多年不见的老朋友（面貌已经变化）仍能一眼认出。

从机器内部的实现方法来看，传统的信息查询是基于传统数字计算机的按地址存取方式进行的；而研究表明，人脑的联想功能是基于神经网络的按内容记忆方式进行的。也就是说，只要是内容相关的事情，不管在哪里（与存储地址无关），都可由其相关的内容被想起。例如，苹果这一概念，一般有形状、大小、颜色等特征，所要介绍的内容记忆方式就是由形状（比如苹果是圆形的）想起颜色、大小等特征，而不需要关心其在人脑中的内部地址。

当前，对机器联想功能的研究中，人们就是利用这种按内容记忆原理，采用一种称为

“联想存储”的技术实现联想功能。联想存储的特点如下：

- (1) 可以存储许多相关（激励，响应）模式对；
- (2) 通过自组织过程可以完成这种存储；
- (3) 以分布、稳健的方式（可能会有很高的冗余度）存储信息；
- (4) 可以根据接收到的相关激励模式产生并输出适当的响应模式；
- (5) 即使输入激励模式失真或不完全时，仍然可以产生正确的响应模式；
- (6) 可在原存储中加入新的存储模式。

联想存储可分为矩阵联想存储、全息联想存储、Walsh 联想存储和网络联想存储等。

3. 机器推理

机器推理就是计算机推理，也称自动推理。它是人工智能的核心课题之一。因为推理是人脑的一个基本功能和重要功能，事实上几乎所有的人工智能领域都与推理有关，因此要实现人工智能，就必须将推理的功能赋予机器，实现机器推理。

所谓推理就是从一些已知判断（称为前提）推出一个新判断（称为结论）的思维过程。在形式逻辑中，推理分为演绎推理、归纳推理和类比推理等基本类型。但这里的机器推理只包括其中的演绎推理。此外，还包括那些非逻辑的基于指令性规则的操作变换式的“推理”。总之，这里的机器推理是广义的推理。

机器推理当然可以模拟人脑推理的宏观方式或过程，即按形式逻辑中的推理规则，用符号演算的方法进行推理（事实上，这是当前机器推理的主要方法），但也可以运用其他方法，例如，数值计算的方法实现推理。

逻辑演绎推理是目前在计算机上实现得较好的一种推理，特别是其中的三段论和假言推理。这一方面是由于这种推理是使用最广的推理形式，另一方面是由于数理逻辑中的一阶谓词逻辑提供了实现这种推理的形式语言，而且这种语言很适合当前的数字计算机。用这种语言，不仅可以在计算机上实现类似于人推理的自然演绎法推理，而且也可实现不同于人的归结（或称消解）法推理。

除了基于经典的二值逻辑的推理外，机器推理还涉及基于各种非经典（或非标准）逻辑的推理，如模态逻辑、时态逻辑、动态逻辑、模糊逻辑、多值逻辑、多类逻辑和非单调逻辑等。基于这些逻辑的推理对人工智能非常重要，但还有许多理论和技术问题需要解决。

从知识表示来看，对于不同的知识表示有不同的推理方式。例如，基于语义网和框架知识表示的推理是一种称为继承的推理。此外，还有常识推理等。

从推理的可靠性来看，推理还可分为精确推理（或确定性推理）和不精确推理（或不确定性推理）。精确推理的前提和结论都是明晰而确定的。传统的逻辑推理都是精确推理。不精确推理的前提和结论则是模糊的、随机的、不完全或不确定的。所以按不确定性的性质来分，不精确推理大体可分为基于概率逻辑的或然推理（如概率推理或统计推理）和基于模糊逻辑的似然推理（一般称为模糊推理）。前者是面向随机性的，后者是面向模糊性的。对于不精确推理，虽然现在已提出了不少推理模型，但这个问题还未得到彻底解决，因此仍是一个需继续研究的课题。

由于受机器硬件的限制等原因，传统的机器推理基本上都是串行推理。为了提高推理速度，并行推理已是当前的一个重要研究方向。事实上，关于并行推理近年来已有不少研

究成果，提出了一些并行推理算法和语言，如并行 Prolog。当然，要实现真正的并行推理，则需有并行计算机硬件的支持。如神经网络计算机，其推理方式就是并行推理，除了传统的符号推理外，现在还发展了许多别的推理技术，如约束推理、定性推理、范例推理、数值化推理、模糊—神经推理等。

综上所述，机器推理是人工智能的基本的、重要的、至今仍在不断发展的技术领域。

4. 机器学习

机器学习就是机器自己获取知识。具体来讲，机器学习主要有这几层意思：

- (1) 对人类已有知识的获取（这类似于人类的书本知识学习）；
- (2) 对客观规律的发现（这类似于人类的科学发现）；
- (3) 对自身行为的修正（这类似于人类的技能训练和对环境的适应）。

按学习方法分类，机器学习一般可分为机械学习、指导学习、解释学习、类比学习、示例学习、发现学习等。这些学习方法都属于所谓的符号学习。除符号学习外，还有连接学习。连接学习就是神经网络学习。它有一般性、鲁棒性、抗噪声等特点，是一种很有前途的机器学习。这就是说，按实现途径分类，机器学习又可分为符号学习和连接学习。当然，这两种途径各有优缺点，因此，将二者结合，使其互补是当前机器学习的一个发展方向。

机器学习是人工智能中的十分重要的研究领域，从 20 世纪 50 年代就开始研究，虽然已取得了不少成就，但仍存在不少困难和问题。

5. 机器理解

机器理解主要包括自然语言理解和图形理解等。

自然语言理解就是计算机理解人类的自然语言，如汉语、英语等，并包括口头语言和文字语言两种形式。试想，计算机如果能理解人类的自然语言，那么计算机的使用将会变得十分方便和简单，而且机器翻译也将真正成为现实。

自然语言理解是一个十分困难的课题，因为人的自然语言本身往往具有二义性，再加上同一句话在不同的时间、地点、场合往往有不同的含义。理解困难的另一个原因是，究竟什么是理解，几乎和什么是智能一样，至今还是一个没有完全明确的问题，因而从不同的角度有不同的解释。从微观来讲，理解是指从自然语言到机器内部表示的一种映射；从宏观来讲，理解是指能够完成所希望的一些功能。例如，美国认知心理学家 G. M. Ullson 曾为理解提出了 4 条判别标准：

- (1) 能够成功地回答与输入材料有关的问题；
- (2) 能够具有对所给材料进行摘要的功能；
- (3) 能用不同的词语叙述所给材料；
- (4) 具有从一种语言转译成另一种语言的能力。

当然，这 4 条标准也只是理解的充分条件，事实上理解也可以表现为某种行为。

图形理解是图形识别的自然延伸，也是计算机视觉的组成部分。对于三维图形的理解称为物景分析。20 世纪 70 年代前，物景分析多限于简单的积木世界。70 年代后，物景分析已进入比较复杂的世界，如识别曲线物体，在复杂的背景中寻找目标以及室外物景

分析等。

由上所述可以看出，理解实际是感知的延伸，或者说是深层次的感知。理解不是对现象或形式的感知，而是对本质和意义的感知。例如，自然语言理解和图形理解都是如此。

6. 机器行为

机器行为主要指机器人行动规划。它是智能机器人的核心技术，规划功能的强弱反映了智能机器人的智能水平。因为，虽然感知能力可使机器人认识对象和环境，但解决问题，还要依靠规划功能拟定行动步骤和动作序列。例如，给定工件装配任务，机器人按照什么步骤去操作每个工件？在杂乱的环境下，机器人如何寻求避免与障碍碰撞的路径，去接近某个目标？机器人规划系统的基本任务是：在一个特定的工作区域中自动地生成从初始状态到目标状态的动作序列、运动路径和轨迹的控制程序。

1.4.3 基于实现技术的领域划分

按照人工智能的实现技术，可以将人工智能的研究领域划分为知识工程与符号处理技术、神经网络技术等。

1. 知识工程与符号处理技术

知识工程是 1977 年美国斯坦福大学的费根鲍姆（E. A. Feigenbaum）教授提出的概念，它涉及知识获取、知识表示、知识管理、知识运用以及知识库与知识库管理系统等一系列知识处理技术。这一技术是以知识为中心的观点来组织智能系统的。

符号处理技术指基于符号演算的推理技术和学习技术，符号处理技术实际是知识工程的基础技术。如前所述，这一领域一直是人工智能的主要研究领域。

2. 神经网络技术

这一领域主要研究各种神经网络模型及其学习算法。这一领域是当前人工智能研究的一个十分活跃且很有前途的分支领域。

1.4.4 基于应用领域的领域划分

按照人工智能的应用领域，可以将人工智能的研究领域划分为问题求解、自动定理证明、自动程序设计、自动翻译、智能控制、智能管理、智能决策、智能通信、智能 CAD、智能 CAI 等。

1. 问题求解

这里的问题，即难题，主要指那些没有算法解，或虽有算法解但在现有机器上无法实施或无法完成的困难问题。例如，路径规划、运输调度、电力调度、地质分析、测量数据解释、天气预报、市场预测、股市分析、疾病诊断、故障诊断、军事指挥、机器人行动规划、机器博弈等。在这些难题中，有些是组合数学理论中所称的 NP（nondeterministic

polynomial, 非确定型多项式) 问题或 NP 完全 (nondeterministic polynomial complete, NPC) 问题。NP 问题是指那些既不能证明其算法复杂性超出多项式界, 但又未找到有效算法的一类问题, 而 NP 完全问题又是 NP 问题中最困难的一种问题。例如, 有人证明过排课表问题就是一个 NP 完全性问题。

游戏世界中的一些智力性问题, 例如, 梵塔问题、农夫过河问题、八数码问题、八皇后问题、旅行商问题、魔方魔圆问题及计算机博弈问题等, 也是一些难题。人工智能也研究这些难题的求解。研究这类难题求解有双重意义: 一方面, 可以找到解决这些难题的途径; 另一方面, 由解决这些难题而发展起来的一些技术和方法可用于人工智能的其他领域。这也正是人工智能研究初期, 研究内容基本上都集中于游戏世界的智力性的重要原因。例如, 博弈问题就可作为搜索策略、机器学习等研究提供很好的实际背景。

2. 自动定理证明

自动定理证明就是机器定理证明, 这也是人工智能的一个重要的研究领域, 也是最早的研究领域之一。定理证明是最典型的逻辑推理问题之一, 它在发展人工智能方法上起过重大作用。如关于谓词演算中推理过程机械化的研究, 帮助人们更清楚地了解到某些机械化推理技术的组成情况。很多非数学领域的任务, 如医疗诊断、信息检索、规划制定和难题求解, 都可以转化成一个定理证明问题。所以自动定理证明的研究具有普遍的意义。

自动定理证明的方法主要有 4 类:

(1) 自然演绎法。它的基本思想是依据推理规则, 从前提和公理中可以推出许多定理, 如果待证的定理恰在其中, 则定理得证。

(2) 判定法。即对一类问题找出统一的计算机上可实现的算法解。在这方面一个著名的成果是我国数学家吴文俊教授 1977 年提出的初等几何定理证明方法。

(3) 定理证明器。它研究一切可判定问题的证明方法。

(4) 计算机辅助证明。它是以计算机为辅助工具, 利用机器的高速度和大容量, 帮助人完成手工证明中难以完成的大量计算、推理和穷举。证明过程中所得到的大量中间结果, 又可以帮助人形成新的思路, 修改原来的判断和证明过程, 这样逐步前进直至定理得证, 这种证明方法的一个重要成果就是, 1976 年 6 月美国的阿佩尔 (K. Appel) 等人合作, 证明了 124 年未能解决的四色定理, 引起了全世界的轰动。一般来讲, 适于计算机辅助证明的是这样一类问题: 它需要检查的信息量极大, 且证明过程需根据中间结果反复由人修改。

3. 自动程序设计

自动程序设计就是让计算机设计程序。具体来讲, 就是人只要给出关于某程序要求的非常高级的描述, 计算机就会自动生成一个能完成这个要求目标的具体程序。所以, 这相当于给机器配置了一个“超级编译系统”, 它能够对高级描述进行处理, 通过规划过程, 生成所需的程序。但这只是自动程序设计的主要内容, 它实际是程序的自动综合。自动程序设计还包括程序自动验证, 即自动证明所设计程序的正确性。这样, 自动程序设计也是人工智能和软件工程相结合的课题。

4. 自动翻译

自动翻译即机器翻译，就是完全用计算机作为两种语言之间的翻译。机器翻译由来已久。早在电子计算机问世不久，就有人提出了机器翻译的设想。随后就开始了这方面的研究。当时人们总以为只要用一部双向词典及一些语法知识就可以实现两种语言文字间的机器互译，结果遇到了挫折。例如，当把“光阴似箭”的英语句子“Time flies like an arrow”翻译成日语，然后再翻译回来的时候，竟变成了“苍蝇喜欢箭”；又如，当把“心有余而力不足”的英语句子“The spirit is willing but the flesh is weak”翻译成俄语，然后再翻译回来时竟变成了“酒是好的，肉变质了”，即“The wine is good but the meat is spoiled”。这些问题的出现才使人们发现，机器翻译并非想像的那么简单，单纯地依靠“查字典”的方法不可能解决翻译问题，只有在对语义理解的基础上，才能做到真正的翻译。所以，机器翻译的真正实现，还要靠自然语言理解方面的突破。

5. 智能控制

智能控制就是把人工智能技术引入控制领域，建立智能控制系统。自从国际知名美籍华裔科学家傅京孙(KS. Fu)在1965年首先提出把人工智能的启发式推理规则用于学习控制系统以来，国内外众多的研究者投身于智能控制研究，并取得一些成果。经过20年努力，到了20世纪80年代中叶，智能控制新学科的形成条件已逐渐成熟。1985年8月，IEEE在美国纽约召开了第一届智能控制学术讨论会。会上集中讨论了智能控制原理和智能控制系统的结构。1987年1月，在美国费城由IEEE控制系统学会和计算机学会联合召开了智能控制国际学术讨论会。会议显示出智能控制的长足进展，也说明了新技术和高技术的发展要求重新考虑自动控制科学及其相关领域，这次会议表明，智能控制已作为一门新学科，出现在国际科学舞台上。

智能控制具有两个显著的特点：第一，智能控制是同时具有知识表示的非数学广义世界模型和传统数学模型混合表示的控制过程，也往往是含有复杂性、不完全性、模糊性或不确定性以及不存在已知算法的过程，并以知识进行推理，以启发来引导求解过程。第二，智能控制的核心在高层控制，即组织级控制，其任务在于对实际环境或过程进行组织，即决策与规划，以实现广义问题求解。

智能控制系统的智能可归纳为以下几方面。

(1) 先验智能：有关控制对象及干扰的先验知识，可以从一开始就考虑到控制系统的设计中；

(2) 反应性智能：在实时监控、辨识及诊断的基础上，对系统及环境变化的正确反应能力；

(3) 优化智能：包括对系统性能的先验性优化及反应性优化；

(4) 组织与协调智能：表现为对并行耦合任务或子系统之间的有效管理与协调。

智能控制的开发，目前认为有以下途径：

(1) 基于专家系统的专家智能控制；

(2) 基于模糊推理和计算的模糊控制；

(3) 基于人工神经网络的神经网络控制；

(4) 综合以上3种方法的综合型智能控制。

6. 智能管理

智能管理就是把人工智能技术引入管理领域,建立智能管理系统。智能管理是现代管理科学技术发展的新动向。智能管理是人工智能与管理科学、系统工程、计算机技术及通信技术等多学科、多技术互相结合、互相渗透而产生的一门新技术、新学科。它研究如何提高计算机管理系统的智能水平,以及智能管理系统的设计理论、方法与实现技术。

智能管理系统是在管理信息系统、办公自动化系统、决策支持系统的功能集成、技术集成的基础上,应用人工智能专家系统、知识工程、模式识别、人工神经网络等方法和技术,进行智能化、集成化、协调化,设计和实现的新一代的计算机管理系统。

7. 智能决策

智能决策就是把人工智能技术引入决策过程,建立智能决策支持系统。智能决策支持系统是在20世纪80年代初提出来的。它是决策支持系统与人工智能,特别是专家系统相结合的产物。它既充分发挥了传统决策支持系统中数值分析的优势,也充分发挥了专家系统中知识及知识处理的特长,既可以进行定量分析,又可以进行定性分析,能有效地解决半结构化和非结构化的问题,从而扩大了决策支持系统的范围,提高了决策支持系统的能力。

智能决策支持系统是在传统决策支持系统的基础上发展起来的,由传统决策支持系统再加上相应的智能部件就构成了智能决策支持系统。智能部件可以有多种模式,例如,专家系统模式、知识库系统模式等。专家系统模式是把专家系统作为智能部件,这是目前比较流行的一种模式。该模式适合于以知识处理为主的问题,但它与决策支持系统的接口实现比较困难。知识库系统模式是以知识库作为智能部件。在这种情况下,决策支持系统就是由模型库、方法库、数据库、知识库组成的四库系统。这种模式接口比较容易实现,其整体性能也较好。

一般来说,智能部件中可以包含如下一些知识:

- (1) 建立决策模型和评价模型的知识;
- (2) 如何形成候选方案的知识;
- (3) 建立评价标准的知识;
- (4) 如何修正候选方案,从而得到更好候选方案的知识;
- (5) 完善数据库,改进对它的操作及维护的知识。

8. 智能通信

智能通信就是把人工智能技术引入通信领域,建立智能通信系统。智能通信就是在通信系统的各个层次和环节上实现智能化。例如,在通信网的网控、转接、信息转换等环节,都可实现智能化。这样,网络就可运行在最佳状态,使呆板的网变成活化的网,使其具有自适应、自组织、自学习、自修复等功能。

9. 智能仿真

智能仿真就是将人工智能技术引入仿真领域,建立智能仿真系统。仿真是对动态模型

的实验,即行为产生器在规定的实验条件下驱动模型,从而产生模型行为。具体地说,仿真是在3种类型知识——描述性知识、目的性知识及处理知识的基础上产生另一种形式的知识——结论性知识。因此可以将仿真看作是一个特殊的知识变换器,从这个意义上讲,人工智能与仿真有着密切的关系。

利用人工智能技术能对整个仿真过程(包括建模、实验运行及结果分析)进行指导,能改善仿真模型的描述能力,在仿真模型中引进知识表示将为研究面向目标的建模语言打下基础,提高仿真工具面向用户、面向问题的能力。从另一方面来讲,仿真与人工智能相结合可使仿真更有效地用于决策,更好地用于分析、设计及评价知识库系统,从而推动人工智能技术的发展。正是基于这些方面,近年来,将人工智能特别是专家系统与仿真相结合,就成为仿真领域中一个十分重要的研究方向,引起了大批仿真专家的关注。

10. 智能 CAD

智能 CAD 简称 ICAD,就是把人工智能技术引入计算机辅助设计领域,建立智能 CAD 系统。事实上,AI 几乎可以应用到 CAD 技术的各个方面。从目前发展的趋势来看,至少有下述4个方面:

- (1) 设计自动化;
- (2) 智能交互;
- (3) 智能图形学;
- (4) 自动数据采集。

从具体技术来看,ICAD 技术大致可分为如下几种方法:

- (1) 规则生成法;
- (2) 约束满足方法;
- (3) 搜索法;
- (4) 知识工程方法;
- (5) 形象思维方法。

11. 智能 CAI

智能 CAI 就是把人工智能技术引入计算机辅助教学领域,建立智能 CAI 系统,即 ICAI。ICAI 的特点是能对学生因材施教地进行指导。为此,ICAI 应具备下列智能特征:

- (1) 自动生成各种问题与练习;
- (2) 根据学生的水平和学习情况自动选择与调整教学内容与进度;
- (3) 在理解教学内容的基础上自动解决问题生成解答;
- (4) 具有自然语言的生成和理解能力;
- (5) 对教学内容有解释咨询能力;
- (6) 能诊断学生错误,分析原因并采取纠正措施;
- (7) 能评价学生的学习行为;
- (8) 能不断地在教学中改善教学策略。

为了实现上述 ICAI 系统,一般把整个系统分成专门知识、教导策略和学生模型等3个基本模块和1个自然语言的智能接口。

总之, ICAI 已是人工智能的一个重要应用领域和研究方向, 引起了人工智能界和教育界的极大关注和共同兴趣。特别是 20 世纪 80 年代以来, 由于知识工程、专家系统技术的进展, 使得 ICAI 与专家系统的关系日益密切。近几届美国与国际人工智能会议都把 ICAI 的研究列入议程, 甚至还召集了专门的智能教学系统会议。

1.4.5 基于应用系统的领域划分

按照应用系统, 可以将人工智能的研究领域划分为专家系统、知识库系统、智能数据库系统、智能机器人系统等。

1. 专家系统

专家系统就是基于人类专家知识的程序系统。专家系统的特点是拥有大量的专家知识(包括领域知识和经验知识), 能模拟专家的思维方式, 面对领域中复杂的实际问题, 能做出专家水平级的决策, 像专家一样解决实际问题。

专家系统的出现, 大大推动了人工智能的发展, 使人工智能真正从实验走向实际, 开始了以知识为中心的人工智能新时代。

2. 知识库系统

所谓知识库系统, 从概念来讲, 它可以泛指所有包含知识库的计算机系统(这是广义理解); 也可以仅指拥有某一领域专门知识以及常识的知识咨询系统(这是一种狭义理解)。按广义理解, 专家系统、智能数据库系统等也都是知识库系统。这里对知识库系统按狭义理解。

与数据库系统类似, 知识库系统的结构包括知识库和知识库管理系统。知识库包括领域的一些基本事实和有关规则等。知识库管理系统负责对知识库的维护、更新和咨询等。这种知识库的咨询不同于数据库的查询, 它不是根据用户的询问, 直接从数据库中检索出有关答案, 而是运用知识库中的知识, 通过推理而间接地得到有关答案。所以, 从某种意义上看, 数据库中存放的是显式的数据, 而知识库中存放的是隐式的数据。另一方面, 从知识库的内容来看, 其基础仍然是数据。所以知识库系统实际上是数据库系统的发展和提高。

事实上, 知识库系统也正是从数据库系统发展演化而来的。例如, 人们给关系数据库加上规则和推理模块, 就可实现所谓的演绎数据库, 而演绎数据库也可看作是一种知识库。

从需求上讲, 知识库系统也是从数据处理到知识处理的必然结果。人们研究知识库, 是从两个方向进行的。一个方向是从 AI 出发, 另一个方向从数据库出发, 二者现在已不谋而合, 交汇于一处了。事实上, 随着数据库理论研究的深入, 知识库已在模型论的基础上, 从经典数据库模型发展到更注重语义联系的、更高抽象层次的概念数据模型。这些抽象模型实质上已经和从 AI 角度提出的若干知识表示方式十分类似。

从数据库出发来研究知识库, 可以从数据库和数据库管理系统中取得借鉴和启发。在这方面目前有两个重要的研究方向: 一个是从面向对象的数据库系统出发来研究面向对象的知识库系统; 另一个是由主动数据库得到启发来研究主动知识库。

3. 智能数据库系统

智能数据库系统就是给传统数据库系统中再加上智能成分。例如，对象数据库、主动数据库等，都是智能数据库系统。

4. 智能机器人系统

智能机器人是这样一类机器人：它能认识工作环境、工作对象及其状态，能根据人给予的指令和“自身”认识外界的结果来独立地决定工作方法，实现任务目标，并能适应工作环境的变化。具体来讲，智能机器人应具备4种机能：感知机能、思维机能、人一机通信机能和运动机能。智能机器人的特征就在于它与外部世界的对象、环境和人相协调的工作机制，智能机器人系统的实现技术，要综合运用人工智能的各种技术，或者说，智能机器人系统是人工智能技术的全面体现和综合运用。

1.4.6 基于计算机系统结构的领域划分

按照计算机系统结构，可以将人工智能的研究领域划分为智能操作系统、智能多媒体系统、智能计算机系统、智能网络系统等。

1. 智能操作系统

智能操作系统就是将人工智能技术引入计算机的操作系统之中，从质上提高操作系统的性能和效率。

智能操作系统的基本模型，将以智能机为基础，并能支撑外层的AI应用程序，以实现多用户的知识处理和并行推理。智能操作系统主要有3大特点：并行性、分布性和智能性。并行性是指能够支持多用户、多进程，同时进行逻辑推理和知识处理；分布性是指把计算机的硬件和软件资源分散而又有联系地组织起来，能支持局域网或远程网处理；智能性又体现于3个方面：一是操作系统所处理的对象是知识对象，具有并行推理和知识操作功能，支持智能应用程序的运行。二是操作系统本身的绝大部分程序也将使用AI程序（规则和事实）编制，充分利用硬件并行推理功能。三是其系统管理应具有较高智能程度的自动管理维护功能，如故障的监控分析等，以帮助系统维护人员做出必要的决策。

2. 智能多媒体系统

多媒体技术是当前计算机最为热门的研究领域之一。多媒体计算机系统就是能综合处理文字、图形、图像和声音等多种媒体信息的计算机系统。智能多媒体就是将人工智能技术引入多媒体系统，使其功能和性能得到进一步发展和提高。事实上，多媒体技术与人工智能所研究的机器感知、机器理解等技术也正好不谋而合。所以，智能多媒体实际上是人工智能与多媒体技术的有机结合。人工智能的计算机视听觉、语音识别与理解、语音对译、信息智能压缩等技术运用于多媒体系统，将会使现在的多媒体系统产生质的飞跃。

3. 智能计算机系统

智能计算机系统就是人们正在研制的新一代计算机系统。这种计算机系统从基本元件到体系结构，从处理对象到编程语言，从使用方法到应用范围，同当前的冯·诺依曼型计算机相比，都有质的飞跃和提高，它将全面支持智能应用开发，且自身就具有智能。

4. 智能网络系统

智能网络系统就是将人工智能技术引入计算机网络系统。如在网络控制、信息检索与转换、人机接口等环节，运用 AI 的技术与成果。研究表明，AI 的模糊技术和神经网络技术可用于网络的连接控制、业务量管制、业务量预测、资源动态分配、业务流量控制、动态路由选择、动态缓冲资源调度等许多方面。

1.4.7 基于实现工具与环境的领域划分

按照实现工具与环境，可以将人工智能的研究领域划分为智能软件工具、智能硬件平台等。

1. 智能软件工具

包括开发建造智能系统的程序语言和工具环境等，这方面现已有不少成果，如函数程序设计语言 LISP、逻辑程序设计语言 Prolog、对象程序设计语言 Smalltalk 与 C++、框架表示语言 FRL、产生式语言 OPS5、神经网络设计语言 AXON、智能体（agent）程序设计语言等，以及各种专家系统工具、知识工程工具、知识库管理系统等。

2. 智能硬件平台

这指直接支持智能系统开发和运行的机器硬件，这方面现在也取得了不少成果。如 LISP 机、Prolog 机、神经网络计算机、知识信息处理机、模糊推理计算机、面向对象计算机、智能计算机等，以及由这些计算机组成的网络系统，有的已研制成功，有的正在研制之中。

1.5 人工智能的基本技术

尽管人工智能还是一门正在探索和发展中的学科，尽管人工智能至今还未形成完整的理论体系，但就其目前各个分支领域的研究内容来看，人工智能的基本技术，基本上应包括以下内容：推理技术、搜索技术、知识表示与知识库技术、归纳技术、联想技术等。

1.5.1 推理技术

几乎所有的人工智能领域都要用到推理，因此，推理技术是人工智能的基本技术之一。需要指出的是，对推理的研究往往涉及对逻辑的研究。逻辑是人脑思维的规律，从而也是

推理的理论基础。机器推理或人工智能用到的逻辑，主要包括经典逻辑中的谓词逻辑和由它经某种扩充、发展而来的各种逻辑。后者通常称为非经典或非标准逻辑。经典逻辑中的谓词逻辑，实际是一种表达能力很强的形式语言。用这种语言不仅可供人用符号演算的方法进行推理，而且也可供计算机用符号推演的方法进行推理。特别是利用一阶谓词逻辑不仅可在机器上进行像人一样的“自然演绎”推理，而且还可以实现不同于人的“归结反演”推理。后一种方法是机器推理或自动推理的主要方法。它是一种完全机械化的推理方法。基于一阶谓词逻辑，人们还开发了著名的逻辑程序设计语言 Prolog。

非标准逻辑是泛指除经典逻辑以外的那些逻辑，如多值逻辑、多类逻辑、模糊逻辑、模态逻辑、时态逻辑、动态逻辑、非单调逻辑等。各种非标准逻辑是为弥补经典逻辑的不足而发展起来的，也可以说是应人工智能之需而发展起来的。例如，为了克服经典逻辑“二值性”的限制，人们发展了多值逻辑及模糊逻辑。实际上，这些非标准逻辑都是由对经典逻辑作某种扩充和发展而来的。在非标准逻辑中，又可分为两种情况，一种是对经典逻辑的语义进行扩充而产生的，如多值逻辑、模糊逻辑等。这些逻辑也可看作是与经典逻辑平行的逻辑。因为它们使用的语言与经典逻辑基本相同，区别在于经典逻辑中的一些定理在这种非标准逻辑中不再成立，而且增加了一些新的概念和定理。另一种是对经典逻辑的语构进行扩充而得到的，如模态逻辑、时态逻辑等。这些逻辑一般都承认经典逻辑的定理，但在两个方面进行了补充，一是扩充了经典逻辑的语言，二是补充了经典逻辑的定理。例如，模态逻辑增加了两个新算子 L （……是必然的）和 M （……是可能的），从而扩大了经典逻辑的词汇表。

上述逻辑为推理特别是机器推理提供了理论基础，同时也开辟了新的推理技术和方法。随着推理的需要，还会出现一些新的逻辑，同时，这些新逻辑也会提供一些新的推理方法。事实上，推理与逻辑是相辅相成的。一方面，推理为逻辑提出课题；另一方面，逻辑为推理奠定基础。

1.5.2 搜索技术

所谓搜索，就是为了达到某一“目标”而连续地进行推理的过程。搜索技术就是对推理进行引导和控制的技术，它也是人工智能的基本技术之一。事实上，许多智能活动的过程，甚至所有智能活动的过程，都可看作或抽象为一个“问题求解”过程，所谓“问题求解”过程，实质上就是在显式的或隐式的问题空间中进行搜索的过程，即在某一状态图，或者与或图，或者一般地说，在某种逻辑网络上进行搜索的过程。例如，难题求解（如旅行商问题）是明显的搜索过程，而定理证明实际上也是搜索过程，它是在定理集合（或空间）上搜索的过程。

搜索技术也是一种规划技术，因为对于有些问题，其解就是由搜索而得到的“路径”。搜索技术是人工智能中发展最早的技术。在人工智能研究的初期，“启发式”搜索算法曾一度是人工智能的核心课题。截至目前，对启发式搜索的研究，人们已取得了不少成果。如著名的 A^* 算法和 AO^* 算法就是两个重要的启发式搜索算法。但是至今，启发式搜索仍然是人工智能的重要研究课题之一。

传统的搜索技术都是基于符号推演方式进行的。近年来，人们又将神经网络技术用于

问题求解，开辟了问题求解与搜索技术研究的新途径。例如，用 Hopfield 网解决 31 个城市的旅行商问题，已取得了很好的效果。

1.5.3 知识表示与知识库技术

知识表示是指知识在计算机中的表示方法和表示形式，它涉及知识的逻辑结构和物理结构。知识库类似于数据库，所以知识库技术包括知识的组织、管理、维护、优化等技术。对知识库的操作要靠知识库管理系统的支持。显然，知识库与知识表示密切相关。

需要说明的是，知识表示实际也隐含着知识的运用，知识表示和知识库是知识运用的基础，同时也与知识的获取密切相关。

对知识表示与知识库的研究，虽然已取得了不少成果，但仍有许多问题需要解决，如知识的分类、知识的一般表示模式、不确定性知识的表示、知识分布表示、知识库的模型、知识库与数据库的关系、知识库管理系统等。

人们都知道“知识就是力量”这句名言。在人工智能的研究中，人们则更进一步领略到了这句话的深刻内涵。的确，对智能来说，知识太重要了，以致可以说“知识就是智能”。因为所谓智能，就是发现规律、运用规律的能力，而规律就是知识。所以，所谓智能也就是发现知识和运用知识的能力，而发现知识和运用知识本身还需要知识。因此，知识是智能的基础和源泉。所以，从这个意义讲，知识表示与知识库是人工智能的核心技术。

1.5.4 归纳技术

归纳技术是指机器自动提取概念、抽取知识、寻找规律的技术。显然，归纳技术与知识获取及机器学习密切相关，因此，它也是人工智能的重要基本技术。

归纳可分为基于符号处理的归纳和基于神经网络的归纳，这两种途径目前都有很大发展。基于神经网络的归纳不必多说（因为神经网络本身就是一个归纳器），值得一提的是基于符号处理的归纳技术。除了已开发出的归纳学习方法外，近年来，基于数据库的数据开采（data mining, DM）和知识发现（knowledge discovery in database, KDD）技术异军突起，方兴未艾，这为归纳技术的发展和应用注入了新的活力。

还需要说明的是，由于归纳时需要分析、综合、比较，还需要反馈、修正、调整和优化等步骤。所以，广义地讲，归纳技术也包括类比、控制、适应甚至进化在内。

1.5.5 联想技术

联想是最基本、最基础的思维活动：它几乎与所有的 AI 技术息息相关。因此，联想技术也是人工智能的一个基本技术。联想的前提是联想记忆或联想存储，这也是一个富有挑战性的技术领域。

以上介绍了人工智能的一些基本理论和技术，因为这些理论和技术仍在不断发展和完善之中，所以它们同时也是人工智能的基本课题。

1.6 人工智能的产生与发展

1.6.1 人工智能学科的产生

1956年夏季,由美国达特莫斯(Dartmouth)大学麦卡锡(J. McCarthy)与哈佛大学的明斯基(M. L. Minsky)、IBM公司信息研究中心的洛切斯特(N. Lochester)、贝尔实验室的香农(C. E. Shannon)共同发起,邀请IBM公司的莫尔(T. More)和塞缪尔(A. L. Samuel)、麻省理工学院的塞尔夫里奇(O. Selfridge)和索罗门夫(R. Solomonff)以及兰德公司和卡内基工科大学的纽厄尔(A. Newell)、西蒙(H. A. Simon)等,共10位来自数学、心理学、神经生理学、信息论和计算机等方面的学者和工程师,在达特莫斯大学召开了一次历时两个月的研讨会,讨论关于机器智能的有关问题。会上经麦卡锡提议正式采用了“人工智能”这一术语。从此,一门新兴的学科便正式诞生了。

需要指出的是,人工智能学科虽然正式诞生于1956年的这次学术研讨会,但实际上它是逻辑学、心理学、计算机科学、脑科学、神经生理学、信息科学等学科发展的必然趋势和必然结果。单就计算机来看,其功能从数值计算到数据处理,再下去必然是知识处理。实际上就其当时的水平而言,也可以说计算机已具有某种智能的成分了。

另一方面,实现人工智能也是人类自古以来的渴望和梦想。据史书《列子·汤问》篇记载,远在公元前九百多年前的我国西周时期,周穆王曾路遇一个名叫偃师的匠人,他献给穆王一个“机器人”,这个“机器人”能走路、唱歌、跳舞,使穆王误以为是一个真人。这虽然是一个传说,但却反映了人类很早就有人工智能的设想。在现代,当电子计算机刚问世不久,天才的英国科学家图灵(A. M. Turing)就于1950年发表了题为“计算机与智能”的论文,推出了著名的“图灵测验”,为人工智能提出了更为明确的设计目标和测试准则。

1.6.2 符号主义学派

1956年之后的10多年间,人工智能的研究取得了许多引人瞩目的成就。从符号主义的研究途径来看,主要成就如下:

(1) 1956年,美国的纽厄尔、肖和西蒙合作编制了一个名为逻辑理论机(logic theory machine, LT)的计算机程序系统。该程序模拟了人用数理逻辑证明定理时的思维规律。利用LT,纽厄尔等人证明了怀特海和罗素的名著——《数学原理》第2章中的38条定理(1963年在另一台机器上证明了全部52条定理)。美籍华人、数理逻辑学家王浩于1958年在IBM-704计算机上用3~5min证明了《数学原理》中有关命题演算的全部定理(220条),并且还证明了谓词演算中150条定理的85%。

(2) 1956年,塞缪尔研制成功了具有自学习、自组织、自适应能力的跳棋程序。这个程序能从棋谱中学习,也能从下棋实践中提高棋艺,1959年它击败了塞缪尔本人,1962年又击败了美国一个州的冠军。

(3) 1959 年, 籍勒洛特发表了证明平面几何问题的程序, 塞尔夫里奇推出了一个模式识别程序; 1965 年, 罗伯特 (Roberts) 编制出了可以分辨积木构造的程序。

(4) 1960 年, 纽厄尔、肖和西蒙等人通过心理学试验总结出了人们求解问题的思维规律, 编制了通用问题求解程序 (general problem solving, GPS)。该程序可以求解 11 种不同类型的问题。

(5) 1960 年, 麦卡锡研制成功了面向人工智能程序设计的表处理语言 LISP。该语言以其独特的符号处理功能, 很快在人工智能界风靡起来。它武装了一代人工智能学者, 至今仍然是人工智能研究的一个有力工具。

(6) 1965 年, 鲁宾逊 (Robinson) 提出了消解原理, 为定理的机器证明做出了突破性的贡献。

在这一时期, 虽然人工智能的研究取得了不少成就, 但就所涉及的问题来看, 大都是一些可以确切定义并具有良好的结构的问题; 就研究的内容来看, 主要集中于问题求解中的搜索策略或算法, 而轻视了与问题有关的领域知识。当时人们朴素地认为, 只要能找到几个推理定律, 就可解决人工智能的所有问题。所以, 这一时期人工智能的研究主要是以推理为中心。因此, 有人将这一时期称为人工智能的推理期。

推理期的人工智能, 基本上是停留在实验室, 没有面向真实世界的复杂问题。此后, 在认真考查了现实世界中的各种复杂问题后, 人们发现要实现人工智能, 除了推理搜索方法之外, 还需要知识。于是人工智能的研究又开始转向知识。

1965 年, 美国斯坦福大学的费根鲍姆 (E. A. Feigenbaum) 教授领导研制的基于领域知识和专家知识的名为 DENDRAL 的程序系统, 标志着人工智能研究的一个新时期的开始。该系统能根据质谱仪的数据并利用有关知识, 推断出有机化合物的分子结构。该系统当时的能力已接近于、甚至超过有关化学专家的水平, 后来在英、美等国得到了实际应用。由于 DENDRAL 系统的特点主要是依靠其所拥有的专家知识解决问题, 因此后来人们称它为专家系统。

继 DENDRAL 之后, 还有一些著名的专家系统, 如医学专家系统 MYCIN、地质勘探专家系统 PROSPECTOR、计算机配置专家系统 R1 等也相继问世。这些专家系统一方面进一步完善了专家系统的理论和技术基础, 同时也扩大了专家系统的应用范围。

由于专家系统走出了实验室, 能解决现实世界中的实际问题, 被誉为“应用人工智能”, 所以, 专家系统很快也就成为人工智能研究中的热门课题, 并受到企业界和政府部门的关注和支持。

在这一时期, 还发生了一些重大学术事件, 如 1969 年, 国际人工智能联合会议 (International Joint Conferences on Artificial Intelligence, IJCAI) 宣告成立。1970 年, 国际性的人工智能专业杂志 *Artificial Intelligence* 创刊。1972 年, 法国马赛大学的科麦瑞尔 (A. Colmerauer) 在 Horn 子句的基础上提出了逻辑程序设计语言 Prolog。

1977 年, 在第五届国际人工智能会议上, 费根鲍姆进一步提出了“知识工程”的概念。这样, 人工智能的研究便从以推理为中心转向以知识为中心, 进入了所谓的知识期。从此以后, 专家系统与知识工程便成为人工智能的一个最重要的分支领域。同时, 知识是智能的基础和源泉的思想也逐渐渗透到人工智能的其他分支领域, 如自然语言理解、景物分析、文字识别和机器翻译等, 从而运用知识 (特别是专家知识) 进行问题求解, 便成为一种新

的潮流。

进入 20 世纪 80 年代后,专家系统与知识工程在理论、技术和应用方面都有了长足的进步和发展。专家系统的建造进入应用高级开发工具时期。专家系统结构和规模也在不断扩大,出现了所谓的多专家系统、大型专家系统、微专家系统和分布式专家系统等。同时,知识表示、不精确推理、机器学习等方面也都取得了重要进展。各个应用领域的专家系统更如雨后春笋般地在世界各地不断涌现。进一步,还出现了不限于专家知识的知识系统和知识库系统。现在,专家系统、知识工程的技术已应用于各种计算机应用系统,出现了智能管理信息系统、智能决策支持系统、智能控制系统、智能 CAD 系统、智能 CAI 系统、智能数据库系统、智能多媒体系统等。

1.6.3 连接主义学派

从连接主义的研究途径看,早在 20 世纪 40 年代,就有一些学者开始了神经元及其数学模型的研究。例如,1943 年,心理学家 McCulloch 和数学家 Pitts 提出了形式神经元的数学模型——现在称为 MP 模型;1944 年,Hebb 提出了改变神经元连接强度的 Hebb 规则。MP 模型和 Hebb 规则至今仍在各种神经网络中起重要作用。

20 世纪 50 年代末到 60 年代初,开始了人工智能意义下的神经网络系统的研究。一批研究者结合生物学和心理学研究的成果,开发出各种神经网络,开始用电子线路实现,后来较多的是用更灵活的计算机模拟,如 1957 年罗圣勃莱特(P. Rosenblatt)开发的称为感知器(perception)单层神经网络、1962 年维特罗(B. Windrow)提出的自适应线性元件(adaline)等。这些神经网络已可用于诸如天气预报、电子线路板分析、人工视觉等许多问题。当时,人们似乎感到智能的关键仅仅是如何构造足够大的神经网络的方法问题。

但这种设想很快就消失了。类似的网络求解问题的失败和成功同时并存,造成无法解释的困扰。人工神经网络开始了一个失败原因的分析阶段。作为人工智能创始人之一的著名学者明斯基(Minsky)应用数学理论对以感知器为代表的简单网络做了深入的分析,结果 1969 年他与白伯脱(Papert)共同发表了颇有影响的 *Perceptions* 一书。书中证明了那时使用的单层人工神经网络,无法实现一个简单的异或门(XOR)所完成的功能。因而明斯基本人也对神经网络的前景持悲观态度。

由于明斯基的理论证明和个人的威望,这本书的影响很大,使许多学者放弃了在该领域中的继续努力,政府机构也改变基金资助的投向。另一方面,由于在此期间,人工智能的基于逻辑与符号推理途径的研究不断取得进展和成功,也掩盖了发展新途径的必要性和迫切性。于是,神经网络的研究进入低谷。

然而,仍有少数几个杰出科学家,如寇耐(T. Kohonen)、葛劳斯伯格(S. Grossberg)、安特生(J. Andenson)等,在极端艰难的环境下仍坚韧不拔地继续努力。

经过这些科学家的艰苦探索,神经网络的理论和技术在经过近 20 年的暗淡时期后终于有了新的突破和惊人的成果。1985 年,美国霍布金斯大学的赛诺斯(T. Seinowsk)开发了名为 NETtalk 英语读音学习用的神经网络处理器,输入为最多由 7 个字母组成的英语单词,输出为其发音。由于该处理器自己可以学习许多发音规则,因此从一无所知起步,经过 3 个月的学习所达到的水平已可同经过 20 年研制成功的语音合成系统相媲美。同年,

美国物理学家霍普菲尔特 (J. Hopfield) 用神经网络迅速求得了巡回推销员路线问题的最优解, 显示它在求解“难解问题”上的非凡能力。实际上, 早在 1962 年, 霍普菲尔特就提出了著名的 HNN 模型。在这个模型中, 他引入了“计算能量函数”的概念, 给出了网络稳定性判据, 从而开拓了神经网络用于联想记忆和优化计算的新途径。此外, 还有不少成功的例子。这些重大突破和成功, 轰动了世界。人们开始对冷落了近 20 年的神经网络又刮目相看了。另一方面, 在这一时期, 虽然在符号主义途径上, 人工智能在专家系统、知识工程等方面取得很大的进展, 但在模拟人的视听觉和学习、适应能力方面, 却遇到了很大的困难。这又使人们不得不回过头来对人工智能的研究途径做新的反思, 不得不寻找新的出路。正是在这样的背景下, 神经网络研究的热潮又再度出现。

1987 年 6 月, 第一届国际神经网络会议 (ICNN) 在美国圣地亚哥召开。会议预定 800 人, 但实际到会达 2000 多人。会上气氛之热烈, 群情之激昂, 据报导是国际学术会议前所未有的。例如, 会上有人竟喊出了“AI is dead. Long live Neural Networks”的口号。会议决定成立国际神经网络学会, 并出版会刊 *Neural Networks*。从此之后, 神经网络便东山再起, 其研究活动的总量急剧增长, 理论与技术继续进展, 应用成果不断涌现, 新的研究机构、实验室、商业公司与日俱增, 世界各国政府也在组织与实施有关的科研攻关项目。现在, 神经网络已经和专家系统知识工程并驾齐驱, 成为人工智能的两个主流方向。

神经网络在智能控制、语音识别与合成、图形文字识别、数据压缩、知识工程、最优化问题求解、智能计算机等领域进行的实践和取得的初步成果, 预示着人工智能的应用不久将会有重大突破。

神经网络学科的发展和应用也迎来了脑神经科学、认知科学、心理学、微电子学、控制论和机器人学、信息技术和数理科学等学科的相互促进、相互发展的空前活跃时期, 特别是在计算机科学研究领域, 将从根本上改变人们传统的数值、模拟、串行、并行、分布等计算与处理概念的内涵和外延, 出现了分布式并行新概念、数值模拟混合的新途径, 探索和开创光学计算机、生物计算机、第 n 代计算机的新构想, 为 21 世纪计算机科学与技术的飞速发展奠定了思想和理论基础。

1.6.4 人工智能的发展趋势

当前人工智能的发展趋势, 主要体现在如下几个方面。

(1) 传统的“符号智能”(传统的基于符号处理技术的人工智能被称为“符号智能”)与“计算智能”(CI), 特别是神经计算, 各取所长, 有机融合。由于两种方法各有所长, 又各有所短, 但却刚好互补。所以, 将两种方法相结合, 则是可取的策略和方向。作为两者结合的智能系统, 神经网络将主要模拟形象思维, 实现识别、联想、学习和适应等功能, 负责对外界环境的感知和交互; 专家系统将主要模拟逻辑思维, 实现判断、推理和搜索等功能, 负责高层的决策与控制。这两种思维方式在人类大脑中结合这样一个事实, 有力地支持着这种发展方向。

(2) 一批新思想、新理论、新技术不断涌现。如模糊技术、模糊-神经网络、遗传算法、进化程序设计、混沌理论、人工生命等, 它们又构成了所谓的“软计算”技术或“计算智能”, 还有“粗糙集 (rough sets)”理论、数据开采 (DM) 与“知识发现 (KDD)”技

术、面向对象技术、现场 AI (situated AI) 等, 而且这些理论和技术又互相渗透、互相融合。

(3) 以 agent (称为“主体”或“智能主体”、“智能体”、“智能代理”等) 概念为基础的分布式人工智能 (distributed artificial intelligence, DAI) 正异军突起, 蓬勃发展。分布式人工智能基于计算机网络, 以研究和开发“群体智能”为主要特征。所以 DAI 与因特网结合将相得益彰。

特别值得指出的是, agent 这一新概念不仅为人工智能注入了新的活力, 开辟了新的方向, 提供了新的思路和技术手段, 而且它也将对整个计算机科学技术产生巨大而深远的影响。特别对软件开发, “面向 agent 技术”将是继面向对象技术之后的又一个突破。事实上, “面向 agent 程序设计”已在因特网上显露头角。

(4) 基于人工智能的应用研究愈加深入而广泛。当今的人工智能研究与实际应用的结合越来越紧密, 受应用的驱动越来越明显。事实上, 现在的人工智能技术已同整个计算机科学技术紧密地结合在一起了, 其应用也与传统的计算机应用越来越融合在一起了。

总之, 以上特点展现了人工智能学科的繁荣景象和光明前途。虽然在通向其最终目标的道路上, 还有不少困难、问题和挑战, 但前进和发展毕竟是大势所趋。

1.6.5 中国人工智能的研究与发展

1977 年, 涂序彦 (原中国人工智能学会理事长) 和郭荣江在《自动化》第 1 期上发表了国内首篇关于 AI 的论文——《智能控制及其应用》, 拉开了中国人工智能研究的序幕。从此, 中国在人工智能方面的研究便蓬勃兴起。

20 世纪 80 年代, 各种人工智能学术团体纷纷成立。1980 年 4 月, 以常迥教授为首的中国自动化学会模式识别与机器智能专业委员会在武汉成立。1980 年 8 月, 以王湘浩教授为首的全国高校人工智能研究会在长春成立。1981 年 9 月, 以秦元勋教授为首的中国人工智能学会在长沙成立。1982 年 4 月, 以王湘浩教授为首的中国计算机学会人工智能学组在杭州成立 (1986 年 11 月, 在太原升级为人工智能与模式识别专业委员会)。1986 年 5 月, 以何志均教授为首的中国软件行业协会人工智能协会在北京成立。1987 年 6 月, 以陆汝钤教授为首的中国计算机学会软件专业委员会智能软件学组在北京成立。1987 年, 国家高技术智能计算机系统专家组在北京成立。这些学术团体和组织经常举办全国性和国际性学术会议, 创办学术期刊, 组织和领导科研攻关, 有力地推动了中国人工智能事业的发展。

近年来, 这些学术组织还每两年召开一次联合学术会议, 并出版了题为《人工智能进展》的会议论文集。

当前, 中国的人工智能研究正方兴未艾, 非常活跃, 并取得了不少举世瞩目的成就。

本章小结

本章介绍人工智能的基本概念、研究目标、基本内容、研究途径与方法、人工智能研究的特点、人工智能的研究领域、人工智能的基本技术、人工智能的产生与发展, 以及发

展趋势等。

人工智能是研究如何模拟、延伸和扩展人的智能,研究与开发各种机器智能和智能机器的理论、方法与技术,涉及计算机科学、脑科学、神经生理学、心理学、语言学、逻辑学、认知(思维)科学、行为科学和数学,以及信息论、控制论和系统论等众多学科领域的一门综合性的交叉学科和边缘学科。它借助于计算机建造智能系统,完成诸如模式识别、自然语言理解、程序自动设计、自动定理证明、机器人、专家系统等智能活动。

人工智能研究的研究目标分为远期目标和近期目标,远期目标是要制造智能机器,近期目标是实现机器智能。

人工智能研究的基本内容是认知建模、机器感知、机器思维、机器学习、机器行为,以及智能系统与智能计算机等。人工智能的研究途径与方法是:基于结构模拟的神经计算、基于功能模拟的符号推演、基于行为模拟的控制进化。人工智能研究的主要特点是:以知识为中心、运用推理技术、启发式搜索、数据驱动方式、用人工智能语言建造系统等。

人工智能的研究领域:在经典的人工智能研究中,分为逻辑推理与定理证明、博弈、自然语言处理、专家系统、自动程序设计、机器学习、人工神经网络、机器人学、模式识别、计算机视觉、智能控制、智能检索、智能调度与指挥、智能决策支持系统、知识发现和数据挖掘,以及分布式人工智能等。按照人工智能的实现技术划分的研究领域有知识工程与符号处理技术、神经网络技术等。按照人工智能的应用领域划分的研究领域有问题求解、自动定理证明、自动程序设计、自动翻译、智能控制、智能管理、智能决策、智能通信、智能 CAD、智能 CAI 等。按照人工智能的应用系统划分的研究领域有专家系统、知识库系统、智能数据库系统、智能机器人系统等。按照计算机系统结构划分的研究领域有智能操作系统、智能多媒体系统、智能计算机系统、智能网络系统等。按照实现工具与环境划分研究领域有智能软件工具、智能硬件平台等。

人工智能的基本技术,基本上应包括以下内容:推理技术、搜索技术、知识表示与知识库技术、归纳技术、联想技术等。

人工智能学派有符号主义(symbolism)、连接主义(connectionism)和行为主义(actionism)等,或者叫做逻辑学派(logicism)、仿生学派(bionicsism)和生理学派(physiologism)。此外还有计算机学派、心理学派和语言学派等。

人工智能学科正式诞生于 1956 年,20 世纪 60 年代符号智能技术获得较大发展,70 年代专家系统等进入实用阶段,80 年代连接主义学派的人工神经网络技术成为研究热点,90 年代人工智能一批新思想、新理论、新技术涌现。21 世纪初,各种人工智能技术出现有机融合和全面发展。

当前人工智能的发展趋势,主要体现在如下几个方面:首先,传统的“符号智能”与“计算智能”,特别是神经计算,各取所长,有机融合。其次,一批新思想、新理论、新技术不断涌现,如:模糊技术、模糊-神经网络、遗传算法、进化程序设计、混沌理论、人工生命、粗糙集(rough sets)理论、数据开采(DM)与知识发现(KDD)技术、面向对象技术、现场 AI(situated AI)等。第三,以 agent 概念为基础的分布式人工智能 DAI 技术正异军突起,蓬勃发展。最后,基于人工智能的应用研究愈加深入而广泛。

习题 1

1. 什么是人工智能？人工智能的意义和目标是什么？
2. 人工智能有哪些研究途径和方法？它们的关系如何？
3. 人工智能研究的基本内容有哪些？
4. 人工智能研究有哪些特点？
5. 人工智能有哪些研究领域？
6. 人工智能有哪些基本技术？
7. 人工智能有哪些主要学派？
8. 简述人工智能的发展概况和当前发展趋势。

第 2 章 知识表示方法

知识是一切智能行为的基础，也是人工智能的重要研究内容。要使计算机具有智能，就必须使它具有知识。适当选择和正确使用知识表示方法可以极大地提高人工智能问题求解的效率。

本章讨论与知识表示相关的问题，包括知识表示的概念和知识表示方法等。在引入知识相关概念的基础上，对现有的多种知识表示方法：一阶谓词逻辑表示法、产生式表示法、语义网络表示法、框架表示法、脚本表示法、过程表示法、Petri 网表示法、面向对象表示法、状态空间表示法、问题归约表示法等进行介绍，分别从它们的基本思想、工作流程、主要特点及相互比较等方面进行分析。

目前，许多学者都积极致力于该方向的研究，并且已经取得了一定的成果。

2.1 知识的基本概念

知识是人们在改造客观世界的实践中积累起来的认识和经验。要认识客观世界，首先应该能够描述客观世界。通常，人们对客观世界的描述是通过数据和信息来实现的。

2.1.1 知识层次

人们描述客观世界的数据、信息、知识等具有如图 2.1 所示的金字塔型层次结构。

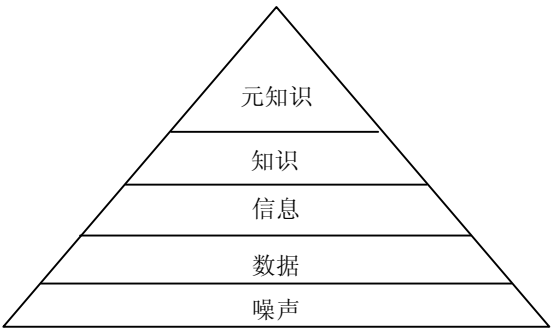


图 2.1 知识的层次结构

图中，底层为噪声，包含不相关或关联性小的数据以及模糊数据；再上一层是数据，它包含潜在关联的数据项；第 3 层是信息，即经过处理的关联数据；再上面就是知识，表示专门信息；最上层则是元知识，它是关于知识的知识。具体的概念解释如下。

(1) 数据

知识处理中的数据不同于通常意义上的“数”，它具有更广泛的含义。例如，符号3，赋予不同的物质就可以代表不同的含义。无论3个人或者3棵树，表达了相同的数量。另外，颜色属性、尺寸大小等都可以作为数据进行记录。这里，数据可以定义为“客观事物的数量、属性、位置及其相互关系等的抽象表示”。

（2）信息

信息是“数据所表示的含义（或称数据的语义）”。如上述同一个数据3在不同的具体场合代表着不同的含义，可以是人、树或是任何其他物质，从而获得不同的解释，产生不同的信息。因此，可以这样来表述数据和信息的关系：数据是信息的载体和表示，而信息是对数据的解释。一般地说，一个信息可用一组描述词及其值来描述：

〈描述词1：值1；…；描述词n：值n〉

它描述一件事、一个物体或一种现象的有关属性、状态、地点、程度、方式等。例如，

〈名字：张三；年龄：50；性别：男；籍贯：北京；文化程度：大专〉

（3）知识

对知识的定义是仁者见仁、智者见智。**Feigenbaum**认为知识是经过整理、加工、解释和转换的信息。**Bernstein**认为知识是由特定领域的描述、关系和过程组成的。**Hayes Roth**则将知识用三维空间来描述，即知识=事实+信念+启发式规则。从知识库的观点看，知识则是某领域中所涉及的各有关方面、状态的一种符号表示。

目前所认可的知识定义是“知识是一个或多个信息的关联”。也就是说，知识是把有关信息关联在一起所形成的信息结构。它反映了客观世界中事物的关系，不同事物或者相同事物间的不同关系形成了不同的知识。知识可以简单表示某一生活常识或规律，也可以表示某种因果关系，前者称为“事实”，后者称为“规则”。例如，“太阳从东方升起”就是一个事实；“如果阴云密布，就可能会下雨”这是一条规则。

（4）元知识

元知识是有关知识的知识，是知识库中的高层知识。包括怎样使用规则、解释规则、校验规则、解释程序结构等知识。元知识存于知识库中，指定可用的数据库资源，并在一个域中确定最为适用的规则组。

2.1.2 知识的属性

知识应具有如下属性：

（1）真伪性

知识是客观事物及客观世界的反映，它具有真伪性，可以通过实践检验其真伪，或用逻辑推理证明其真伪。

（2）相对性

一般知识不存在绝对的真假，都是具有相对性的。在一定条件下或特定时刻为真的知识，当时间、条件或环境发生变化时可能变成假的。

（3）不完全性

知识往往是不完全的。这里的不完全大致分为条件不完全和结论不完全两大类。

（4）不确定性，即模糊性和不精确性

现实中知识的真与假，往往并不总是“非真即假”，可能处于某种中间状态，即所谓具有真与假之间的某个“真度”，即模糊度和不精确度。例如“人老了就可能糊涂”，“老了”、“可能”和“糊涂”都是一些模糊概念。在知识处理中必须用模糊数学或统计方法等来处理模糊的或不精确的知识。

(5) 可表示性

知识作为人类经验存在于人脑之中，虽然不是一种物质东西，但可以用各种方法表示出来。一般表示方式包括符号表示法、图形表示法和物理表示法。

(6) 可存储性、可传递性和可处理性

既然知识可以表示出来，那么就可以把它存储起来；知识既可以通过书本来传递，也可以通过教师的讲授来传播，还可以通过计算机网络等来传输，知识可以从一种表示形式转换为另一种表示形式；知识一旦表示出来，就可以同数据一样进行处理。

(7) 相容性

相容性是关于知识集合的一个属性。即存在于一体（如专家系统的知识库）的所有知识之间应该是相互不矛盾的，从这些知识出发，不能推出相互矛盾的命题。

2.1.3 知识分类

关于知识的类型，可以从不同角度来划分。

从知识的确定程度来分，知识可分为确定性知识和不确定性知识两类。确定性知识是可以给出其真值为“真”或“假”的知识。这些知识是可以精确表示的知识，即可以用经典逻辑命题（有惟一真或假的陈述语句）来描述，是一类“非真即假”的知识。不确定性知识是指具有“不确定”特性的知识。不确定性的概念包含不精确、不完备和模糊。若知识并非“非真即假”，可能处于某种中间状态，这类知识往往要用模糊命题或模态命题来表达。

知识按其性质分，可分为概念、命题、公理、定理、规则和方法等。

知识按其含义分，大致可分为事实、规则、规律、推理方法。事实是对客观事物属性的值的描述。一般这种知识中不含任何变量，可以用一个值为“真”的命题来表达。规则是可分解为前提和结论两部分的用以表达因果关系的知识，一般形式为“如果 A 则 B ”，其中 A 表示前提， B 表示结论。规则还可进一步分为不带变量的规则和带变量的规则两种。规律是事物之间的内在的必然联系。

知识按其作用分，可分为事实性知识、过程性知识和控制性知识。事实性知识（也称为叙述性知识）是用来描述问题或事物的概念、属性、状态、环境及条件等情况的知识。例如，“北京是中华人民共和国的首都”、“鸟有两只翅膀”、“小李正在上计算机课”等，这些都是事实性知识。事实性知识主要反映事物的静态特征，一般采用直接表达形式。过程性知识是用来描述问题求解过程所需要的操作、演算或行为等规律性的知识，它指出在问题求解过程中如何使用那些与问题有关的事实性知识，即用来说明在那些叙述性知识成立的时候该怎么办。过程性知识一般由与所求解问题有关的规则、定律、定理及经验来构成。其表示方法主要有产生式规则、语义网络等。控制性知识（也称无知识或超知识）是关于如何运用已有知识进行问题求解的知识，因此，也称为关于知识的知识。例如，问题

求解中的推理策略（如正向推理、逆向推理）、搜索策略（如广度优先、深度优先、启发式）和不确定性的传播策略等。

知识按其应用范围可分为常识性知识和领域性知识。常识性知识是指通用通识的知识，即人们普遍知道的、适应于所有领域的知识，包括与领域问题求解有关的已被接受的定义、事实和各种理论方法。领域性知识是指面向某个具体专业的专业性知识，这些知识只有该领域的专业人员才能够掌握和运用它。例如，领域专家的经验等。专家系统解决问题主要依靠的就是领域知识。

知识按其在问题求解过程中的作用可分为静态知识和动态知识两类。静态知识主要针对对象性知识，是关于问题领域内事物的事实、关系等，它包括事物的概念、事物的分类、事物的描述等。动态知识是关于问题求解的知识，它常常是一种过程，说明怎样操作已有的数据和静态知识以达到问题的求解，是反映动作过程的过程，如一个问题领域内关于推理路径的方向、推理过程、可理解性等方面的知识、启发性方法等。

按知识的层次性可分为表层知识和深层知识。表层知识是指客观事物的现象以及这些现象与结论之间关系的知识。例如，经验性知识、感性知识等。表层知识形式简洁、易表达、易理解，但它并不能反映事物的本质。目前大多专家系统拥有的知识都是表层知识。深层知识是指事物本质、因果关系内涵、基本原理之类的知识，例如，理论知识、理性知识等。

知识按其等级性可分为零级知识、一级知识和二级知识三个层次。零级知识是常识性知识和原理性知识，是关于问题领域的事实、定理、方程、实验对象和操作等。一级知识是经验性知识，这是由于零级知识对求解不良结构问题常常失灵，因而出现的启发性方法，如单凭经验的规则，含义模糊的建议，不确切的判断标准等。二级知识是运用上述两级知识的知识。这种知识层次还可以继续划分下去，把零级知识和一级知识称为领域（或目标）知识，把二级以上知识称为元知识。高级知识对低级知识有指导意义。在专家系统设计中，领域知识是必不可少的。

通常把元知识分为两类。一类是人们知道的知识，这类知识刻画了领域知识的内容和结构的一般特征，如知识产生的背景、范围、可信程度等；另一类是关于如何运用知识的知识，如在问题求解当中所采用的推理方法，为解决一个特殊任务而必须完成的活动的计划、组织和选择方面的知识。近年来，元知识的开发与运用逐渐引起了人们的重视，它是提高专家系统性能的一种有效途径，已成为新一代专家系统的一个重要标志。

2.1.4 知识表示

一个智能系统的智能性很大程度上取决于知识的数量及其可利用的程度。系统中可利用的知识越多，其智能性就可能越高。知识表示就是研究和解决如何将所需要的知识用适当的形式表示出来并存放放到计算机中，即将知识表示成为计算机可以接受的形式。

1. 知识表示的概念

所谓知识表示就是知识的符号化和形式化的过程，是研究用机器表示知识的可行性、有效性的一般方法，是一种数据结构与控制结构的统一体，既考虑知识的存储又考虑知识

的使用。知识表示可以看成是一组描述事物的约定，以把人类知识表示成机器能处理的数据结构。

知识表示方法研究各种数据结构的设计，通过这种数据结构把问题领域的各种知识结合到计算机系统的程序设计过程。一般来说，对于同一种知识可以采用不同的表示方法，反过来，一种知识表示方法可以表达多种不同的知识。然而，在求解某一问题时，不同的表示方法会产生完全不同的效果。迄今为止，人们还没有找到一种通用、完善的知识表示模式，知识表示还没有完善的理论可循。

2. 知识表示方法的分类

对知识表示方法的研究离不开对知识的研究与认识。由于目前对人类知识的结构及其机制还没有完全搞清楚，因此关于知识表示的理论及其规范尚未建立起来。尽管如此，人们在对智能系统的研究及其建立过程中，还是结合具体研究提出了一些知识表示方法。概括起来，这些方法可以分为如下两大类：符号表示法和连接机制表示法。

符号表示法是用各种包含具体含义的符号，以各种不同的方式和次序组合起来表示知识的一类方法，它主要用来表示逻辑性知识。连接机制表示法是用神经网络技术表示知识的一种方法，它把各种物理对象以不同的方式及次序连接起来，并在其间互相传递及加工各种包含具体意义的信息，以此来表示相关的概念及知识。相对于符号表示法而言，连接机制表示法是一种隐式的表示知识方法，它特别适用于表示各种形象性的知识。

另外，按照控制性知识的组织方式进行分类，表示法可分为说明性表示法和过程性表示法。说明性表示法着重于对知识的静态方面，如客体、事件、事实及其相互关系和状态等，其控制性知识包含在控制系统中；过程性表示法强调的是对知识的利用，着重于知识的动态方面，其控制性知识全部嵌入于对知识的描述中，且将知识包含在若干过程之中。

目前，用得较多的知识表示方法主要有一阶谓词逻辑表示法、产生式表示法、框架表示法、语义网络表示法、脚本表示法、过程表示法、Petri 网表示法、面向对象表示法等。

3. 知识表示方法的衡量及特性

既然有诸多的知识表示方法，那么怎样的方法才是合理有效的呢？好的知识表示方法又应当具备怎样的特性呢？下面对此进行讨论。

建立一种知识表示方法，首先要求有较强的表达能力和足够的精细度。其次，相应于表示方法的推理要保证正确性和效率。从使用者观点看，常常希望满足可读性好、模块性好等要求。因此从综合的角度来看，一个好的知识表示应具备以下特性：

（1）完备性

要求具有表达领域问题所需的各种知识的能力，即要求所采用的知识表示方法具有语法完备性和语义完备性，并便于知识库的检查与调试。目前的大多数知识表示方法都很难满足这一要求。由于专门知识、知识库的特点及建库方法所造成的原因，如果不选择表示能力强的方法，就很难使知识库具有某些有关的甚至是很重要的知识，严重影响专家系统

的问题求解能力。

（2）一致性

要求知识库中的知识必须具有一致性，不能相互产生矛盾。几乎所有的专家系统的研制者在开发自己的系统时，都在追求这个目标。由于专家的知识大多是启发性知识，具有不完全性和不确定性。因此，所采用的知识表示必须便于系统进行一致性检查，以便在使用中完善知识库，保证系统的求解质量。

（3）正确性

知识表示必须能真实地反映知识的实际内涵，而不允许有偏差。只有这样，才能保证系统得出正确结论和合理建议。

（4）灵活性

针对不同的专业领域，应当根据具体知识的特点及其自然结构的制约选用不同的知识表示方法。或是用单一方法，或是用混合方法，甚至设计研究新的表示方法，一定要具体问题具体分析，灵活掌握，切忌生搬硬套。

（5）可扩充性

高性能知识库应当不需要做硬件上或控制结构上的修改就能对知识库进行扩充，即要求知识表示模式与运用知识的推理机制相互独立，在专家系统中一般采用知识库与推理机分离的手段来实现这一目的。另一方面，往往专家不能很快地把领域问题的所有知识定义为一个完整的知识库，通常先定义一个子集，不断增加、修改、删除来扩充和完善知识库，这种方法主张将专家系统的知识作为一个开放集来处理，并尽可能模块化地存储知识条目，便于知识库的扩充。

（6）可理解性

知识表示的可理解性指它表示的知识易于被人们理解的程度。易理解的表示模式的好处是显而易见的，它符合人们的思维习惯，便于知识库研制人员把专家的专门知识整理并形式化，也便于知识库的设计、实现和改进。

（7）可利用性

知识表示的目的在于知识的利用，具体地说就是知识的检索和推理。知识的检索与推理是一种控制知识，在专家系统中，一旦知识表示模式被选定，它们也就相应地被确定下来。因此，所选择的知识表示方法应当便于对知识的利用，其数据结构应力求简单，并保持清晰一致。如果一种表示模式的数据结构过于复杂或者难于理解，使得推理不便于进行匹配、冲突消解以及不确定性的计算等处理，那么就势必影响智能系统的效率及其问题求解的能力。

（8）可维护性

知识需要进行合理的组织。对于知识的组织是与其表示方法密切相关的，不同的表示方法对应于不同的组织方式。这就要求在设计和选择知识表示方法时，充分考虑将要对知识进行的组织方式。此外，知识还需要适当地增补、修改和删除，以保证知识的一致性和完整性，即需要进行知识的管理和维护。因此在选择知识表示方法时还应当充分考虑到知识管理和维护的方便性。

简而言之，在建造具体专家系统时，应当以有效地表示问题领域的专门知识，便于知识的获取，有利于运用知识进行推理的原则来选择知识表示方法。

2.2 一阶谓词逻辑表示法

逻辑表示法是一种基于数理逻辑的知识表示方式。数理逻辑是一门研究推理的科学，在人工智能研究中占有重要的地位。人工智能中用到的逻辑可分为两大类：一类是经典命题逻辑和一阶谓词逻辑；另一类是除经典逻辑以外的那些逻辑。

谓词逻辑的表现方式与人类自然语言比较接近，能够自然而精确地表达人类思维和推理的有关知识，因此谓词逻辑已成为各种智能系统中最基本的知识表达方法。

利用逻辑公式，人们能描述对象、性质、状况和关系。使用逻辑法表示知识，将以自然语言描述的知识，通过引入谓词、函数来加以描述，获得有关的逻辑公式，进而将其用内部代码表示。在逻辑法表示下可采用归结法或其他方法进行准确的推理。逻辑表示通常包括命题逻辑和谓词逻辑。关于命题逻辑表示将在第3章做具体介绍，这里主要指的是谓词逻辑表示法。

谓词逻辑表示法采用谓词合式公式 **WFF** 和一阶谓词演算把要解决的问题变为一个有待证明的问题，然后采用消解定理和消解反演来证明一个新语句是从已知的正确语句导出的，从而证明这个新语句也是正确的。谓词逻辑是一种形式语言，能够把数学中的逻辑证明符号化。

2.2.1 命题与真值

定义 2.1 一个陈述句称为一个断言。凡有真假意义的断言称为命题。

命题的意义通常称为真值，它只有真假两种情况。当命题的意义为真时，则称该命题的真值为真，记为 **T**；反之，则称该命题的真值为假，记为 **F**。在命题逻辑中，命题通常用大写的英文字母来表示。

一个命题不能同时既为真又为假。例如，“天安门城楼在长安街的北边”是一个真值为 **T** 的命题，“天安门广场在长安街的北边”则是一个真值为 **F** 的命题。

一个命题可在一定条件下为真，在另一种条件下为假。例如，命题“北京今天有雨”，需要根据当天的实际情况来决定其真值。

没有真假意义的感叹句、疑问句等都不是命题。例如，“今天好冷啊！”和“今天的温度有多少度？”等都不是命题。

命题的优点是简单、明确；其主要缺点是无法描述客观事物的结构及其逻辑特征，也无法表示不同事物间的共性。

2.2.2 论域和谓词

论域是由所讨论对象的全体构成的非空集合。论域中的元素称为个体，论域也常称为个体域。例如，整数的个体域是由所有整数构成的集合，每个整数都是该个体域中的一个个体。

在谓词逻辑中,命题是用谓词来表示的。一个谓词可分为谓词名和个体两部分。其中,个体是命题中的主语,用来表示某个独立存在的事物或者某个抽象的概念;谓词名是命题的谓语,用来表示个体的性质、状态或个体之间的关系等。例如,对于命题“王坚是学生”可用谓词表示为 $\text{STUDENT}(\text{Wangjian})$ 。其中, Wangjian 是个体,代表王坚。 STUDENT 是谓词名,说明王坚是学生的这一特征。通常,谓词名用大写英文字母表示,个体用小写英文字母表示。

谓词可形式地定义如下:

定义 2.2 设 D 是个体域, $P: D^n \rightarrow \{T, F\}$ 是一个映射, 其中

$$D^n = \{(x_1, x_2, \dots, x_n) \mid x_1, x_2, \dots, x_n \in D\}$$

则称 P 是一个 n 元谓词 ($n=1, 2, \dots$), 记为 $P(x_1, x_2, \dots, x_n)$ 。其中 x_1, x_2, \dots, x_n 为个体变元。

在谓词中,个体可以是常量、变元或函数。例如,“ $x > 6$ ”可用谓词表示为 $\text{Greater}(x, 6)$, 其中 x 是变元。再如“王坚的父亲是教师”, 可用谓词表示为 $\text{TEACHER}(\text{father}(\text{Wangjian}))$, 其中 $\text{father}(\text{Wangjian})$ 是一个函数。

函数可形式地定义如下:

定义 2.3 设 D 是个体域, $f: D^n \rightarrow D$ 是一个映射, 则称 f 是 D 上的一个 n 元函数, 记作

$$f(x_1, x_2, \dots, x_n) \quad (n=1, 2, \dots)$$

其中 x_1, x_2, \dots, x_n 为个体变元。

谓词与函数从形式上看很相似,容易混淆,但是它们是两个完全不同的概念。谓词的真值是“真”或“假”,而函数无真假可言,函数的值是个体域中的某个个体。谓词实现的是从个体域中的个体到 T 或 F 的映射,而函数所实现的是在同一个个体域中从一个个体到另一个个体的映射。在谓词逻辑中,函数本身不能单独使用,它必须嵌入到谓词之中。

在谓词 $P(x_1, x_2, \dots, x_n)$ 中,如果 $x_i (i=1, 2, \dots, n)$ 都是个体常量、变元或函数,称它为一阶谓词。如果 x_i 本身又是一个一阶谓词,则称它为二阶谓词。本书仅讨论一阶谓词。

2.2.3 谓词公式与量词

在一阶谓词演算中,合法的表达式称为合式公式,即谓词公式。

当一个谓词公式含有量词时,区分个体变元是否受量词的约束很重要。量词通常有两个,即全称量词 \forall 和存在量词 \exists 。

全称量词 \forall 表示所有的。全称量词符号 \forall 解释为“对于每一个”、“对于所有”等。例如,对于个体域中所有个体 x , 谓词 $F(x)$ 均成立,可用含有全称量词的谓词表示为

$$(\forall x) F(x)$$

存在量词 \exists 表示存在某些。存在量词符号 \exists 解释为“存在”、“至少一个”、“对某些”、“有一个”、“有些”等。例如,若存在某些个体 x 使谓词 $F(x)$ 成立时,可用含有存在量词的谓词表示为

$$(\exists x) F(x)$$

利用上述符号,可把单个谓词组合成复杂的谓词公式,表达复杂的领域知识。

2.2.4 谓词逻辑表示方法

谓词逻辑不仅可以用来表示事物的状态、属性、概念等事实性知识，也可以用来表示事物的因果关系，即规则。对事实性知识，通常是用否定、析取或合取符号连接起来的谓词公式表示。对事物间的因果关系，通常用蕴含式表示，例如，对“如果 x ，则 y ”可表示为“ $x \rightarrow y$ ”。

当用谓词逻辑表示知识时，首先需要根据所表示的知识定义谓词，然后再用连接词或量词把这些谓词连结起来，形成一个谓词公式。

例1 用谓词逻辑表示知识“每个人都有一个父亲”。

首先定义谓词， $\text{PERSON}(x)$ ：表示 x 是人。

$\text{HASFATHER}(x, y)$ ：表示 x 有父亲 y 。

此时，该知识可用谓词表示为

$$(\forall x)(\exists y)(\text{PERSON}(x) \rightarrow \text{HASFATHER}(x, y))$$

例2 用谓词逻辑表示知识“所有教师都有自己的学生”。

首先定义谓词， $\text{TEACHER}(x)$ ：表示 x 是教师。

$\text{STUDENT}(y)$ ：表示 y 是学生。

$\text{TEACHES}(x, y)$ ：表示 x 是 y 的老师。

此时，该知识可用谓词表示为

$$(\forall x)(\exists y)(\text{TEACHER}(x) \rightarrow \text{TEACHES}(x, y) \wedge \text{STUDENT}(y))$$

该谓词公式可读作：对所有 x ，如果 x 是一个教师，那么一定存在一个个体 y ， x 是 y 的老师，且 y 是一个学生。

例3 用谓词逻辑表示知识“所有的整数不是偶数就是奇数”。

首先定义谓词， $\text{I}(x)$ ： x 是整数。

$\text{E}(x)$ ： x 是偶数。

$\text{O}(x)$ ： x 是奇数。

此时，该知识可用谓词表示为

$$(\forall x)(\text{I}(x) \rightarrow \text{E}(x) \vee \text{O}(x))$$

例4 用谓词逻辑表示如下知识：

王涛是计算机系的一名学生。

赵晔是王涛的同班同学。

凡是计算机系的学生都喜欢编程。

首先定义谓词， $\text{COMPUTER}(x)$ ：表示 x 是计算机系的学生。

$\text{CLASSMATE}(x, y)$ ：表示 x 是 y 的同班同学。

$\text{LIKE}(x, y)$ ：表示 x 喜欢 y 。

此时，可用谓词公式把上述知识表示为

$\text{COMPUTER}(\text{Wangtao})$

$\text{CLASSMATE}(\text{Zhaoeye}, \text{Wangtao})$

$$(\forall x)(\text{COMPUTER}(x) \rightarrow \text{LIKE}(x, \text{programming}))$$

逻辑成为最早和使用最广的知识表示方法的主要原因有如下 3 点：

(1) 它具有两个重要的相互关联的部分。一个公理系统，它说明什么样的关系和蕴含可以形式化。一个演绎结构，即推理规则集合，它能从公理集合中推导出定理，如常用的假言三段论、概括、特指等。

(2) 逻辑及其形式系统有如下重要特征：所有演绎都保证正确，而其他方法目前尚难达到这种程度。逻辑语句的集合（从各个原始语句集开始，导出所有结论的闭包集合）的语义保持完整，由推理规则说明。原则上，它可以保证知识库逻辑上的一致性和所有结论的正确性。

(3) 演绎可以完全机械化。程序可以从现有语句中自动确定知识库中某一新语句的有效性。它是自动定理证明中使用得较为成功的一种技术。

形式逻辑根据为真的事实进行推理演算，从而得到新的事实。用逻辑方法求解一个问题的全过程如下：

- (1) 用谓词演算将问题形式化。
- (2) 在这种逻辑表示的形式上建立控制系统。
- (3) 证明从初始状态可以达到目标状态。

2.2.5 谓词逻辑表示方法的 BNF 描述

这里将一阶谓词逻辑的知识表示法用 BNF (Backus normal form) 归纳描述如下：

〈知识〉 ::= { 〈谓词表达式〉 }

〈谓词表达式〉 ::= 〈蕴含式〉 | [量词] (〈变量〉 {, 〈变量〉}) (〈蕴含式〉)

〈蕴含式〉 ::= 〈或式〉 | 〈或式〉 → 〈或式〉

〈或式〉 ::= 〈与式〉 | 〈或式〉 ∨ 〈与式〉

〈与式〉 ::= 〈文字〉 | 〈与式〉 ∧ 〈文字〉

〈文字〉 ::= 〈原子〉 | ∼ 〈原子〉

〈原子〉 ::= 〈谓词〉 [(〈变量〉 {, 〈变量〉})] | 〈谓词〉 [(〈常量〉 {, 〈常量〉})]

〈量词〉 ::= ∀ | ∃

〈变量〉 ::= 〈变量名〉

〈常量〉 ::= 〈个体名〉 | 〈常数〉

用这种表达知识的方法，建造知识库的基本步骤如下：

- (1) 获取并理解领域知识；
- (2) 用自然语句描述领域知识；
- (3) 使用适当的谓词公式表达自然语句；
- (4) 构造 WFF，使其与被表达的自然语句在逻辑上保持一致。

2.2.6 谓词逻辑表示方法的特点

逻辑表示法的主要优点如下：

- ① 符号简单，描述易于理解。

- ② 自然、严密、灵活、模块化。
- ③ 具有严格的形式定义。
- ④ 每项事实仅需表示一次。
- ⑤ 具有证明过程中所使用的推理规则。
- ⑥ 利用定理证明技术可以从旧的事实推出新的事实。

其主要缺点如下：

- ① 难于表示过程式和启发式知识。
- ② 由于缺乏组织原则，利用该方法表示的知识库难于管理。

③ 由于是弱证明过程，当事实的数目增大时，在证明过程中决定使用哪条规则时可能产生组合爆炸。

- ④ 不具有表示不精确和不确定知识的能力。

谓词逻辑法常与其他表示方法混合使用，灵活方便，可以表示比较复杂的问题。当然，一阶逻辑的表达能力也是有限的，如具有归纳结构的知识、多层次的知识类型都难于用一阶逻辑来描述。

2.3 产生式表示法

产生式表示法又称规则表示法，是目前人工智能领域中应用最多的一种知识表示方法，也是一种比较成熟的表示方法。

2.3.1 产生式

产生式一词，首先是由美国数学家 E. Post 提出来的。Post 根据替换规则提出了一种称为波斯特机的计算模型，模型中的每条规则当时被称为一个产生式。后来，这一术语几经修改扩充，被用到许多领域。例如，形式语言中的文法规则也称为产生式。产生式又称为产生式规则，简称规则。

1. 产生式的一般形式

产生式通常用于表示具有因果关系的知识，其一般形式为

前件 \rightarrow 后件

其中，前件为前提，后件为结论或动作。前件和后件可以由逻辑运算符 AND, OR, NOT 组成的表达式。产生式规则的语义是：如果前提满足，则可得结论或者执行相应的动作，即后件由前件来触发。所以，前件是规则的执行条件，后件是规则体。例如：

IF 动物是哺乳动物 AND 吃肉 THEN 该动物是食肉动物
就是一个产生式。

产生式用 BNF 范式严格描述为如下形式：

$\langle \text{产生式} \rangle ::= \langle \text{前件} \rangle \rightarrow \langle \text{后件} \rangle$

$\langle \text{前件} \rangle ::= \langle \text{简单条件} \rangle \mid \langle \text{复合条件} \rangle$

$\langle \text{后件} \rangle ::= \langle \text{事实} \rangle \mid \langle \text{操作} \rangle$

$\langle \text{复合条件} \rangle ::= \langle \text{简单条件} \rangle \text{ AND } \langle \text{简单条件} \rangle [(\text{AND} \langle \text{简单条件} \rangle) \cdots]$
 $\mid \langle \text{简单条件} \rangle \text{ OR } \langle \text{简单条件} \rangle [(\text{OR} \langle \text{简单条件} \rangle) \cdots]$

$\langle \text{操作} \rangle ::= \langle \text{操作名} \rangle [(\langle \text{变元} \rangle, \cdots)]$

2. 产生式与蕴含式

由产生式的一般描述可以看出,产生式与逻辑蕴含式在形式上非常相似,但二者又有所不同。事实上,逻辑蕴含式只是产生式的一种特殊情况。

产生式除逻辑蕴含式外,还包括各种操作、规则、变换、算子、函数等。概括来讲,产生式描述了事物之间的一种对应关系(包括因果关系和蕴含关系),其外延十分广泛。状态转换规则和问题变换规则都是产生式规则;程序设计语言的文法规则、逻辑中的逻辑蕴含式和等价式、数学中的微分和积分公式、化学中分子结构式的分解变换规则等,也都是产生式规则;甚至体育比赛中的规则、国家的法律条文、单位的规章制度等,也都可以表示成产生式规则。

一个产生式规则就是一条知识,它既可以是精确知识又可以是不精确知识。谓词逻辑蕴含式只能表示精确知识。此外,用产生式表示知识的系统中,知识的检验是通过事实与前提条件的匹配来实现的,这样的匹配可以是精确的也可以是不精确的。然而,对于谓词逻辑的蕴含式而言,该匹配必须是精确的。

2.3.2 产生式系统

产生式表示法一般用于所谓的产生式系统。产生式系统最早由 E. Post 于 1943 年提出,并由 A. Newell 和 E. A. Simon 于 1972 年作为一种人类认识模型引入到人工智能研究领域。现在,产生式系统已经在人工智能中广泛应用,并取得了很大进展。

这里讨论的产生式系统,是指一种基于产生式规则表示的知识系统。关于产生式系统的进一步讨论,将在第 3 章进行。

1. 产生式系统的组成

产生式系统是用来描述若干个不同的以产生式规则或产生式条件及其操作为基础、相互配合、协同作用的系统。

产生式系统一般由全局数据库、产生式规则库和控制策略 3 部分组成。

产生式规则库是某领域知识(规则)的存储器,是描述该领域知识的产生式集合。这些规则可以表示为与或树形式。该规则库是产生式系统进行问题求解的基础。它是否能够有效地表达领域内的过程性知识,能否对知识进行合理的组织和管理,直接关系到系统的性能和运行效率。

全局数据库也称为上下文、黑板、事实数据库等。它是一个用于存放问题求解过程中各种当前信息的数据结构。全局数据库中的已知事实通常用字符串、向量、集合、矩阵、表等数据结构表示。当产生式规则库中某条产生式的前提与全局数据库中的某些事实相匹配时,该产生式就被激活,并把用它推出的结论放入全局数据库中,作为后面推理的已知

事实。全局数据库的内容是动态变化的。

控制策略负责规则的选取和系统的运行。它主要完成匹配、冲突消解和操作。具体地讲就是按照一定的策略从产生式规则库中选择规则与全局数据库中的已知事实进行匹配。如果匹配成功的规则不止一条,则需要进行冲突的消解以选取出其中的一条来执行。执行该条规则的过程中,将相应的结论加入全局数据库或者执行指定操作,直至问题得到求解。

2. 产生式系统的分类

产生式系统从不同的角度出发,有不同的分类。比如,按推理方向划分可分为前向、后向和双向产生式系统;按产生式规则库和全局数据库的性质及结构特征划分可分为可交换、可分解、可恢复产生式系统等。

按推理方向划分:

(1) 前向推理,是从已知事实出发,通过规则库求得结论。也被称为数据驱动方式或者自底向上的方式。

(2) 后向推理,是从目标(作为假设)出发,反向使用规则求得已知事实,也称目标驱动方式或者自顶向下的方式。

(3) 双向推理,既自底向上、又自顶向下做双向推理,直至某个中间界面上两方向结果相符便成功结束。这种双向推理较前向推理或后向推理所形成的推理网络要小,从而推理效率较高。运用不同的推理机制就形成了不同的产生式系统。

按组织结构特征划分:

(1) 对于规则的使用次序是可交换的,无论使用哪条规则都可以达到目的的产生式系统称为可交换产生式系统。该系统的特点是搜索过程不必进行回溯,无须记录可用规则的作用顺序,求解效率较高。但是系统要求每条规则的执行都要为全局数据库添加新的内容,这一要求往往难以适用。

(2) 把一个规模较大且比较复杂的问题分解为若干个规模较小且比较简单的子问题分别进行求解的系统称为可分解产生式系统。该系统由于将初始数据库分解为若干子库,每个子库又可再进一步进行分解,从而缩小了搜索范围,提高了问题求解效率。

(3) 在问题求解过程中,当出现求解无法继续的情况时,能够撤销由前一执行规则所产生的结果,使全局数据库恢复到先前的状态,然后选用别的规则继续求解,这样的系统称为可恢复产生式系统。该系统既可以向全局数据库添加新的内容,又可以删除和修改旧的内容,操作灵活方便。

3. 产生式知识表示

产生式表示法容易描述事实、规则以及它们的不确定性度量。

事实可看成是断言一个语言变量的值或是多个语言变量间的关系的陈述句。语言变量的值或语言变量间的关系可以是一个词,不一定是数字。如雪是白色的,其中雪是语言变量,其值是白色的。莉莉喜欢牡丹花,其中莉莉和牡丹花是两个语言变量,两者关系的值是喜欢。

一般情况下,使用三元组(对象,属性,值)表示事实,用(关系,对象1,对象2)来表示对象间关系,若考虑不确定性则可用加入规则强度的四元组进行表示。这种表示及

其内部实现就是一个表。例如，事实小王年龄是 15 岁，可记为 (Wang Age 15)；小王、小李是同学，表达了二者之间的关系，可记为 (Classmate Wang Lee)。

为求解过程查找的方便，在知识库中可将某类有关事实以网状、树状结构组织联结在一起。对于规则，表示事务间的因果关系，以“IF Condition THEN Action”的单一形式来描述，将规则作为知识的单位。其中的 Condition 部分称作条件式前件或模式，而 Action 部分称作动作、后件或结论。条件部分常是一些事实 A_i 的合集，而结论常是某一事实 B，如果考虑不确定性，需另附可信度度量值。

如 MYCIN 系统中一条规则的表示。从专家那里获得的规则是

IF (1) the stain of organism is gramnegative

AND (2) the morphology of the organism is rod,

AND (3) the aerobicity of the organism is anaerobic

THEN there is suggestive evidence (0.6) that the identity of the organism is bacteroides

而用 LISP 实现的机器内部表示为

premise (and (same cntext gram gramneg)

(same cntext morph rod)

(same cntext air anaerobic))

action (conclude cntext zident bacteroides tally 0.6)

为便于规则的使用，在知识库中某些规则常按某种观点组织起来放在一起。

4. 产生式系统的运行流程

产生式系统的运行是一个从初始事实出发，寻求到达目标条件的搜索求解过程。产生式系统求解问题的一般流程如图 2.2 所示。

(1) 对全局数据库进行初始化。即把问题的初始已知事实存入全局数据库。

(2) 判断规则库中是否存在尚未使用过的规则，如果存在且可与全局数据库中的已知事实相匹配，则转第 (3) 步；否则，不存在这样的事实转第 (6) 步。

(3) 执行当前选中的规则，并对该规则做一标记，把该规则执行后所得的结论送入全局数据库。如果该规则的结论部分指出的是某些操作，则执行相应的操作。

(4) 检查全局数据库中是否已经包含了问题的解，如果包含则问题求解结束；否则，转第 (2) 步。

(5) 要求用户提供进一步的问题相关事实，若能提供则转第 (2) 步；否则，问题求解结束。

(6) 规则库中已经没有尚未使用过的规则，问题求解结束。

上述流程只是产生式系统运行的一般简化过程。实际上，问题求解的过程是与控制策略密切相关的，它需要解决诸如冲突消解、不确定性处理等很多具体问题。

动物识别系统是一个典型的用于分析的产生式系统。下面就以它为例做介绍。

这是一个对老虎、猎豹、长颈鹿、斑马、鸵鸟、企鹅、鹰 7 种动物进行辨识的产生式系统。它的规则库如下所示。

R₁: IF 该动物有毛发

THEN 它是哺乳动物

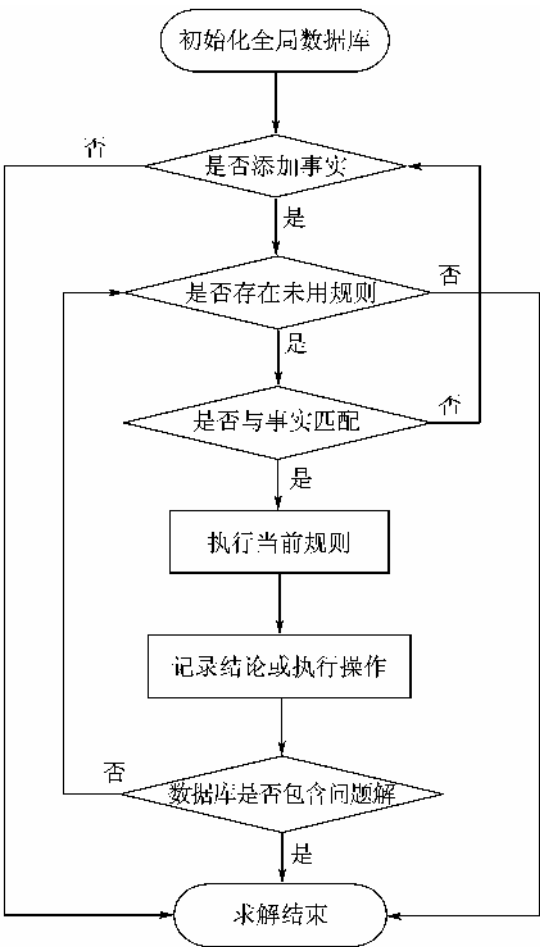


图 2.2 产生式系统运行流程图

- R₂:

IF

该动物能产乳
- THEN

它是哺乳动物
- R₃:

IF

该动物有羽毛
- THEN

它是鸟类
- R₄:

IF

该动物会飞行
- AND

会产蛋
- THEN

它是鸟类
- R₅:

IF

该动物是哺乳动物
- AND

它吃肉
- THEN

它是食肉动物
- R₆:

IF

该动物是哺乳动物
- AND

它长有爪子
- AND

它长有利齿
- AND

它眼睛前视

	THEN	它是食肉动物
R ₇ :	IF	该动物是哺乳动物
	AND	它有蹄
	THEN	它是有蹄类动物
R ₈ :	IF	该动物是哺乳动物
	AND	它反刍
	THEN	它是偶蹄动物
R ₉ :	IF	该动物是食肉动物
	AND	它的体色为黄褐色
	AND	它的身上有黑色条纹
	THEN	它是老虎
R ₁₀ :	IF	该动物是食肉动物
	AND	它的体色为黄褐色
	AND	它的身上有深色斑点
	THEN	它是猎豹
R ₁₁ :	IF	该动物是有蹄类动物
	AND	它长有长腿
	AND	它长有长颈
	AND	它体色为黄褐色
	AND	它的身上有深色斑点
	THEN	它是长颈鹿
R ₁₂ :	IF	该动物是有蹄类动物
	AND	它体色为白色
	AND	它的身上有黑色条纹
	THEN	它是斑马
R ₁₃ :	IF	该动物是鸟类
	AND	它不会飞
	AND	它长有长腿
	AND	它长有长颈
	AND	它体色为黑白相杂
	THEN	它是鸵鸟
R ₁₄ :	IF	该动物是鸟类
	AND	它不会飞
	AND	它会游泳
	AND	它体色为黑白相杂
	THEN	它是企鹅
R ₁₅ :	IF	该动物是鸟类
	AND	它善于飞行
	THEN	它是鹰

上述规则中, $R_1 \sim R_4$ 是对哺乳动物和鸟类的界定; $R_5 \sim R_8$ 是将哺乳动物进一步划分为食肉动物和有蹄类动物; $R_9 \sim R_{10}$ 和 $R_{11} \sim R_{12}$ 分别是对食肉动物和有蹄类动物进行的细分; $R_{13} \sim R_{14}$ 则是对鸟类的划分和进一步细分。

假设有一种动物是鸵鸟, 现在要通过上面的规则库对这一假设进行检验, 以正向推理为例, 首先获得已知事实: 该动物长有长颈, 长有长腿。对照规则库, R_{11} 和 R_{13} 都包含这两个条件, 因而无法确定。申请获得新的事实, 得到“它有羽毛”。这个条件与 R_3 相匹配, 因而可得知该动物为鸟类。再对比 R_{11} 和 R_{13} , 显然与 R_{13} 匹配。这时, 进一步获得事实, “它不会飞”和“它黑白杂色”, 这时所有事实与规则 R_{13} 完全匹配, 从而得到结论“它是鸵鸟”。当然, 该问题也可以通过反向推理来解决, 这里就不赘述了。

2.3.3 产生式表示法的特点

产生式表示格式固定, 形式单一, 规则(知识单位)间相互较为独立, 没有直接关系, 使数据库的建立较为容易, 处理较为简单的问题是可取的。另外推理方式单纯, 也没有复杂的计算。特别是知识库与推理机是分离的, 这种结构给知识库的修改带来方便, 无须修改程序, 对系统的推理路径也容易做出解释。此外, 产生式表示既可以表示确定性知识又可以表示不确定性知识, 既便于表示启发性知识又便于表达过程性知识, 所以产生式表示经常作为建造专家系统的首选知识表示方法。

但是产生式方法也存在着一定的不足。如前所述, 产生式系统的求解过程是一个担负进行“匹配—冲突消解—执行”的过程。由于规则库一般都比较庞大, 匹配通常是十分耗时的, 这样系统的工作效率就受到影响。同时在求解复杂问题时还容易引起组合爆炸。此外, 产生式适合表达具有因果关系的过程性知识, 而对于具有结构关系的知识却是无能为力的。因此, 产生式表示法更适合于表示那些相关性不强、不存在结构关系的领域性知识。

2.3.4 产生式表示法与其他知识表示方法的比较

(1) 产生式表示法与逻辑表示法的比较

产生式表示法的主要思想是基于产生式的, 其描述形式与逻辑蕴含式十分相似。如前所述, 逻辑蕴含式只是产生式的一种特殊形式。然而产生式表示法较之逻辑表示法却有一个突出的优势, 就是可以表示不确定性知识, 因此它能更加广泛地应用。

(2) 产生式表示法与过程表示法的比较

产生式系统也可以看成是一种过程表示方式。两者的主要区别在于: 过程表示允许子程序之间可以直接通信; 在产生式系统中, 产生式规则只能通过全局数据库互相作用。

(3) 产生式表示法与框架表示法的比较

产生式表示法主要用于描述事物之间的因果关系, 而框架表示法则主要用于描述事物的内部结构以及事物之间的类属关系。

2.4 语义网络表示法

语义网络是 Quillian 作为人类联想记忆的一个显式心理学模型提出的。他认为，人在进行交际时总是首先有什么东西需要说，而所说出的全部句子的形式是由想说什么这种意图决定的。因此，他主张在处理文句生成的问题时，需把语义放在首位。Simon 于 1970 年正式提出了语义网络的概念，并讨论了它和一阶谓词的关系。

2.4.1 语义网络的基本结构

语义网络也称为联想网络，是知识表示中最重要的方法之一。语义网络利用结点和带标记的边构成有向图描述事件、概念、状况、动作以及客体之间的关系。带标记的有向图能十分自然地描述客体之间的关系。

语义网络通常由语法、结构、过程和语义 4 部分组成。其中，语法部分决定表示词汇表中允许用哪些符号，它涉及各个结点和弧线；结构部分叙述符号排列的约束条件，指定各弧线连接的结点对；过程部分说明访问过程，这些过程能用来建立和修正描述，回答相应问题；语义部分确定与描述相关的意义和方法，即确定有关结点的排列及其占有物和对应弧线。

语义网络是对知识的有向图表示方法。一个语义网络是由一些以有向图表示的三元图（结点 A ，弧，结点 B ）连接而成。图中，结点表示概念、事物、时间、情况等。弧是有方向和有标注的。方向体现主次，结点 A 为主，结点 B 为辅。弧上的标注表示结点的属性或结点之间的关系。这样的 3 元组如图 2.3 所示。

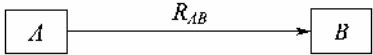


图 2.3 最简单的语义网络（基本网元）

从逻辑表示法来看，一个语义网络相当于一组二元谓词。因为三元组（结点 A ，弧，结点 B ）可写成 P （个体 A ，个体 B ），其中个体 A ，个体 B 对应结点 A 和结点 B ，而弧及其上标注的结点 A 与结点 B 的关系由谓词 P 来体现。

语义网络的 BNF 描述如下：

<语义网络> ::= <基本网元> | Merge (<基本网元>, ...)

<基本网元> ::= <结点> <语义联系> <结点>

<结点> ::= (<属性—值对>, ...)

<属性—值对> ::= <属性名>: <属性值>

<语义联系> ::= <预定义语义联系> | <自定义语义联系>

其中，Merge (...) 表示一个合并过程，它把括弧中的所有基本网元关联在一起，从而构成一个语义网络。图 2.4 就是一个由基本网元合并而成的语义网络结构的示例。

2.4.2 语义网络的知识表示

语义网络同其他知识表示方法一样，应该能够有效地表示事实以及事物间关系两方面

的知识。这两种类型的知识表示在实质上是是一致的，只是表示关系的连接上方的标记不同而已。常识一般用如图 2.5 的方式进行表示，事物之间的关系则是采用所谓的语义联系来表示。

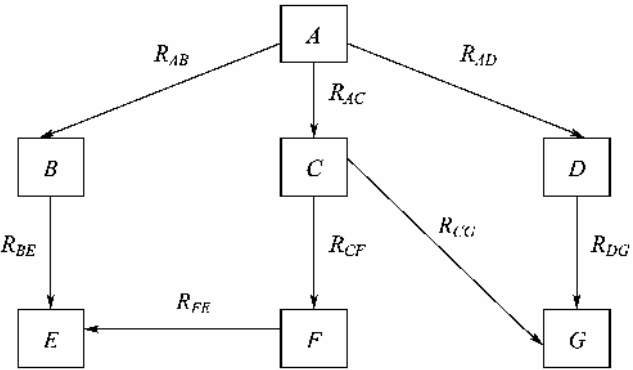


图 2.4 语义网络结构示例

语义网络较之普通网络最大的特点就在于它能表示事物之间的各种关系。因此，在语义网络中关系显得尤为重要，它提供了组织知识的基本结构。没有这些关系，知识就只是基本事实的一个简单集合；有了这些关系的描述，知识就成为可以联系其他知识的关联结构。这些可表示的不同关系统称为语义联系。

常用的语义联系有 ISA (Is-a)，AKO (A-kind-of)，HAVE，AMO (A-member-of) 等，表示事物的性质、属性。此外，还有 Composed-of 表示构成关系；Before，After，At 表示时间关系；Located-on，Located-at，Located-under 等表示事物间的位置关系；Similar-to，Near-to 表示相似或接近关系等。图 2.6 是一个简单的用语义联系表示的例子。

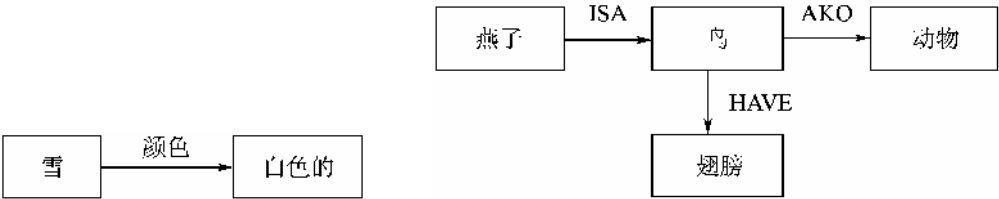


图 2.5 常识表示示例

图 2.6 语义联系示例

在这里要特别提到的是一种常用的知识表述结构，即“对象-属性-值 (object-attribute-value tripe, OAV)”。对象、属性、值这 3 个因素是在构建语义网络过程中最频繁出现的项，它们足以描述一个简化的语义网络。因此，OAV 常常用于提取语义网络的主要特征项，并以列表的形式列举出有关知识，然后通过规则推理转换为机器代码。下面给出一个 OAV 描述的例子，如表 2.1 所示。

表 2.1 OAV 语义结构举例

对象	属性	值	对象	属性	值
狗	毛色	棕黄色	猫	毛色	白色
狗	品种	京巴	猫	品种	波斯
狗	年龄	3	猫	年龄	5

为了更形象地说明语义网络如何通过语义联系表示知识，下面举出一个植物分类语义网络的例子。该网络用命题描述如下：

- (1) 树和草都是植物；
- (2) 树和草都是有根有叶的；
- (3) 水草是草，且长在水中；
- (4) 果树是树，且会结果；
- (5) 苹果树是果树中的一种，它结苹果；
- (6) 芒果树也是果树的一种，它结芒果。

这些命题所构成的语义网络如图 2.7 所示，图中反映了植物属种的继承关系及属性特征。

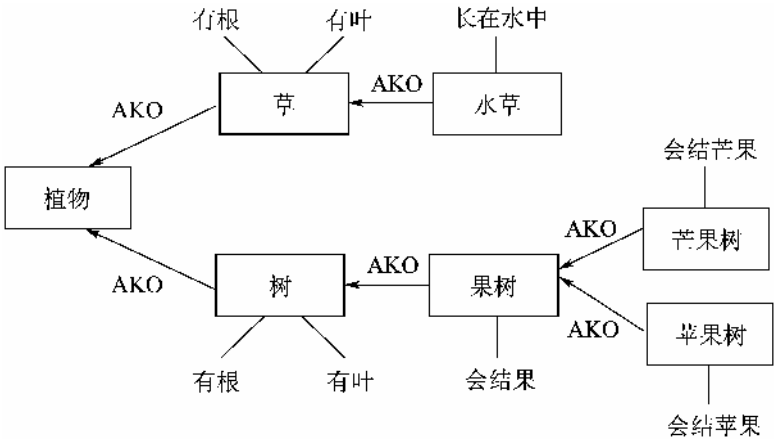


图 2.7 植物分类语义网络

2.4.3 语义网络与 Prolog

语义网络表示很容易转换为 Prolog 语言。因为，Prolog 同样可以有效地表达事实和规则这两方面知识。下面对 Prolog 与语义网络表示之间的对应关系进行介绍。

带有参数的谓词可以表示易于理解的某一事实。例如，

color (red); 红色

father_of (Tom, John); Tom 是 John 的父亲

此外，通过 ISA，HAS-A 等关系所表达的谓词，也可以很好地表示某种二元关系，这正与语义网络的特点相吻合，例如，

Is_a (red, color); 红色是一种颜色

Has_a (John, father); John 有一位父亲

Has_a (John, parents); John 有双亲

但是，对于上例中用 HAS-A 所表示的双亲关系，如果想要知道 John 的父亲或母亲的具体姓名，这样的表示就显得不够了。因此需要在原有谓词之上引入附加谓词来间接表示，例如，

Is_a (Tom, father); Tom 是一位父亲

Is_a (Rose, mother); Rose 是一位母亲

可是还是无法确定 Tom 就是 John 的父亲或者 Rose 就是 John 的母亲,即究竟谁是 John 的双亲。因此, Prolog 采用规则子句来进行描述。形式如下:

$P:-P_1, P_2, L, P_n$

其中, P 为规则头, P_i 为子目标; $:-$ 符号表示“如果”关系。这样前面所说的“双亲”的例子, 就可以用 Prolog 的子句表示为

parent(X, Y) :- father(X, Y)

parent(X, Y) :- mother(X, Y)

就具体例子而言, 有

parent(Tom, John) :- father(Tom, John)

parent(Rose, John) :- mother(Rose, John)

显然, Tom 是 John 的父亲, 因此是 John 的双亲; Rose 是 John 的母亲, 也是 John 的双亲。这样就表达了更为明确的二元关系。甚至还可以表达更进一步的亲缘关系。例如,

grandparent(X, Y) :- parent(X, Z), parent(Z, Y)

子句中, X 与 Y , Z 与 Y 的父子关系, 推导出 X 与 Y 的祖孙关系。

由上可见, Prolog 谓词、谓词表达式以及子句可以很好地与语义网络表示相对应, 它能够有效表示一定的事实以及事物之间的关系。

2.4.4 语义网络的求解流程

语义网络对问题求解是通过两种推理过程, 即继承和匹配来完成的。

继承是指把对事物的描述从概念结点传递到实例结点。概念结点是指表示通用概念、总体名称的结点, 而实例结点是指表示那些相对于一般性概念的具体实例的结点。继承过程分为 3 类: 值继承、If-need 继承以及 Default 继承。匹配也是一种有效的推理方式, 在语义网络中是指通过建立必要的虚结点和虚链, 通过部件匹配来实现问题求解。这里不做具体介绍。

用语义网络求解问题的一般过程如下:

(1) 根据待求解问题的要求构造一个网络片断, 其中有些结点或弧的表示为空, 用来反映待求解的问题。

(2) 依据此网络片断到知识库中去寻找可匹配的网络, 以找出所需要的信息。这种匹配一般不是完全的, 具有不确定性, 因此需要解决不确定性匹配问题。

(3) 当问题的语义网络片断与知识库中的某一语义网络片断相匹配时, 则与该询问相匹配的事实就是所求问题的解。

2.4.5 基本的语义关系

从功能上讲, 语义网络可以描述任何事物间的任意复杂关系。但是, 这种描述是通过把许多基本的语义关系关联到一起来实现的。基本语义关系是构成复杂语义关系的基础,

也是语义网络知识表示的基础。由于基本语义关系的多样性和灵活性,下面给出的仅是一些最常用的基本语义关系。

(1) 类属关系

类属关系是指具有共同属性的不同事物间的分类关系、成员关系或实例关系。它体现的是“具体与抽象”、“个体与集体”的概念。类属关系的一个最主要特征是属性的继承性,处在具体层的结点可以继承抽象层结点的所有属性。常用的类属关系如下:

A_kind_of: 含义为“是一种”,表示一个事物是另一个事物的一种类型。

A_member_of: 含义为“是一员”,表示一个事物是另一个事物的一个成员。

Is_a: 含义为“是一个”,表示一个事物是另一个事物的一个实例。

例如,分类关系“鸟是一种动物”可用 **A_kind_of** 语义关系来表示。它说明鸟是动物的一种类型,并可继承动物的所有属性。又如,成员关系“张强是共青团员”可用 **A_member_of** 语义关系来表示。

在类属关系中,具体层结点除具有抽象层结点的所有属性外,还可以增加一些自己的个性,甚至还能够对抽象层结点的某些属性加以更改。例如,所有的动物都具有能运动、会吃等属性。鸟类作为动物的一种,除具有动物的这些属性外,还具有会飞、有翅膀等个性。

(2) 包含关系

包含关系也称为聚类关系,是指具有组织或结构特征的“部分与整体”之间的关系。它和类属关系的最主要区别是包含关系一般不具备属性的继承性。常用的包含关系如下。

Part_of: 含义为“是一部分”,表示一个事物是另一个事物的一部分。

例如,“大脑是人的一部分”可用 **Part_of** 语义关系来表示。再如,“黑板是墙的一部分”也可用 **Part_of** 来表示。对于这两个例子,从继承性的角度看,大脑不一定具有人的各种属性,黑板也不具有墙的各种属性。

(3) 属性关系

属性关系是指事物和其属性之间的关系。常用的属性关系如下。

Have: 含义为“有”,表示一个结点具有另一个结点所描述的属性。

Can: 含义是“能”、“会”,表示一个结点能做另一个结点的事情。

例如,“鸟有翅膀”可用 **Have** 语义关系来表示。

(4) 时间关系

时间关系是指不同事件在其发生时间方面的先后次序关系。常用的时间关系如下。

Before: 含义为“在前”,表示一个事件在另一个事件之前发生。

After: 含义为“在后”,表示一个事件在另一个事件之后发生。

At: 表示某一事件发生的时间。

例如,“澳门回归在香港回归之后”可用 **After** 语义关系来表示。

(5) 位置关系

位置关系是指不同事物在位置方面的关系。常用的位置关系如下。

Located_on: 含义为“在上”,表示某一物体在另一物体之上。

Located_at: 含义为“在”,表示某一物体所在的位置。

Located_under: 含义为“在下”,表示某一物体在另一物体之下。

Located_inside: 含义为“在内”,表示某一物体在另一物体之内。

Located_outside: 含义为“在外”，表示某一物体在另一物体之外。

例如，“书在桌子上”可用 **Located_on** 语义关系来表示。

(6) 相近关系

相近关系是指不同事物在形状、内容等方面相似或接近。常用的相近关系如下。

Similar_to: 含义为“相似”，表示某一事物与另一事物相似。

Near_to: 含义为“接近”，表示某一事物与另一事物接近。

例如，“猫似虎”可用 **Similar_to** 语义关系来表示。

(7) 组成关系

组成关系表示“构成”联系，是一种一对多的联系，被它联系的结点间不具有属性的继承性。常用的组成关系如下。

Composed_of: 含义为“构成”，表示一个事物由另一些事物所组成。

例如，对于“整数由正整数、负整数及零组成”可用 **Composed_of** 语义关系表示。

(8) 推论关系

推论关系是指从一个概念推出另一个概念的语义关系。常用的组成关系如下。

Infer: 含义为“推出”，表示前提与结论之间的推理关系。

例如，“由成绩好推出学习努力”可用推论语义关系 **Infer** 来表示：“ $A \text{ Infer } B$ ”。这里 A 表示“成绩好”， B 表示“学习努力”。

2.4.6 语义网络表示法的特点

语义网络由于其自然性而被广泛使用。语义网络的主要优点如下：

(1) 重要相关性能被明确地清晰地表示出来。

(2) 基于联想记忆模型，相关事实可以从其直接相连的结点中推导出来，而不必遍历整个庞大的知识库，从而避免了组合爆炸。

(3) 具有继承性，并易于对继承层次进行演绎。

(4) 能够利用少量的基本概念的记忆建立状态和动作的描述。

语义网络也存在一些缺点，表现如下：

(1) 不能保证网络操作所得结论的有效性。

(2) 逻辑表达不充分，无法嵌入启发式信息。

(3) 对于网络不存在标准的术语和约定，语义解释取决于操作网络的程序。

(4) 网络的搜索需要强有力的组织原则。

因此，网络表示法比较合适的领域大多数是根据非常复杂的分类进行推理的领域，以及需要表示事件状况、性质及动作之间关系的领域，特别是在二元关系的表示上突显其优势。不过语义网络表示法并不是一种通用的知识表示方法。

2.4.7 语义网络法与其他知识表示方法的比较

逻辑和产生式表示方法常用于表示有关论域中各个不同状态间的关系，然而用于表示一个事物同其各个部分间的分类知识就不方便了，而槽和填槽表示方法便于表示这种分类

知识。这种表示方法包括框架、概念从属、脚本和语义网络。框架表示将一个特殊个体类同其所有有关断言联结起来。概念从属表示将一个动作或一个事件同其所有相关断言联结起来。脚本表示将一个特殊的时间序列同其所有有关断言联结起来。语义网络是最简单的，它是这类表示法的先驱，同一阶逻辑有相同的表达能力。

虽然语义网络表示法和框架表示法同属一种结构化的表示方法，它能把事物的属性以及事物间的各种语义联系显式地表示出来，下层概念结点可以继承、补充、变异上层概念的属性，从而实现信息的共享。但是二者还是有所区别的。框架表示法适合于表达固定的、典型的概念、事件和行为，而语义网络表示法具有更大的灵活性，用其他表示法能表达的知识几乎都可以用语义网络表示出来。如果把一种事物、概念或情况作为语义网络中的结点，并且用语义联系表示这些结点间的宏观关系，那么每个结点的内部结构关系可用框架来表示。

2.5 框架表示法

框架理论是美国著名学者 Minsky 于 1975 年提出的。该理论认为人们对现实世界中各种事物的认识都是以一种类似于框架的结构存储在记忆当中的，当面临一个新事物时，就从记忆中找出一个适合的框架，并根据实际情况对其细节加以修改补充，从而形成对当前事物的认识。框架表示法正是以框架理论为基础发展起来的一种结构化的知识表示法。目前，已成为一种被广泛使用的知识表示方法。

2.5.1 框架的基本结构

框架是一种集事物各方面属性的描述为一体，并反映相关事物间各种关系的数据结构。它是知识表示的基本单位。一个框架由若干个“槽”构成，槽又依据具体情况划分为若干个“侧面”。槽用于描述对象的某一方面的属性，侧面用于描述相应属性的某一方面。槽与侧面所具有的属性值分别称为槽值和侧面值。在一个用框架表示知识的系统中，一般都含有多个框架，为了指称和区分不同的框架以及一个框架内的不同槽、不同侧面，需要分别给它们取不同的名字，分别称为框架名、槽名及侧面名。无论是对于框架还是槽或侧面，都可以为其附加上一些说明性的信息，一般是指一些约束条件，用于指出什么样的值才能填入槽或侧面中去。一个框架可以有任意有限数目的槽，一个槽可以有任意有限数目的侧面，一个侧面又可以有任意有限数目的侧面值。这样就可以形成一个完整的具有继承性的框架网络。

框架的基本结构可表示如下：

<框架名>

<槽名 1>:	<槽值 1>	<侧面名 11>	值 111, 值 112, ...
		<侧面名 12>	值 121, 值 122, ...

...

<槽名 2>:	<槽值 2>	<侧面名 21>	值 211, 值 212, ...
---------	--------	----------	-------------------

<侧面名 22> 值 221, 值 222, ...

...

<槽名 n>: <槽值 n> <侧面名 n1> 值 n11, 值 n12, ...

<侧面名 n2> 值 n21, 值 n22, ...

...

<约束>: <约束条件 1>

<约束条件 2>

...

<约束条件 k>

即一个框架一般有若干个槽，一个槽有一个槽值或者有若干个侧面，而一个侧面又有若干个侧面值。其中，槽值和侧面值可以是数值、字符串、布尔值，也可以是一个动作或过程，甚至还可以是另一个框架的名字。

例 5 下面是一个描述“教师”的框架。

框架名: <教师>
类属: <知识分子>
工作: 范围: (教学, 科研)
默认: 教学
性别: (男, 女)
学历: (中师, 高师)
类型: (<小学教师>, <中学教师>, <大学教师>)

可以看出，这个框架的名字为“教师”，它含有 5 个槽，槽名分别是“类属”、“工作”、“性别”、“学历”和“类型”。这些槽名的右面就是其值，如“<知识分子>”、“男”、“女”、“高师”、“中师”等。其中，“<知识分子>”又是一个框架名，“范围”、“默认”就是侧面名，其后是侧面值，如“教学”、“科研”等。另外，用尖括号<>括住的槽值也是框架名。

例 6 下面是一个描述“大学教师”的框架。

框架名: <大学教师>
类属: <教师>
学位: (学士, 硕士, 博士)
专业: <学科专业>
职称: (助教, 讲师, 副教授, 教授)
外语: 语种: 范围: (英, 法, 日, 俄, 德, ...)
默认: 英
水平: (优, 良, 中, 差)
默认: 良

例 7 下面是描述一个具体教师的框架。

框架名: <教师-1>
类属: <大学教师>
姓名: 李明
性别: 男

年龄：25

职业：教师

职称：助教

专业：计算机应用

部门：计算机系软件教研室

工作：参加工作时间：2002年8月

工龄：当前年份-参加工作年份

工资：<工资单>

比较例6和例7中的框架可以看出，前者描述的是一个概念，后者描述的则是一个具体的事物。二者的关系是，后者是前者的一个实例。因此，后者一般称为前者的实例框架。这就是说，这两个框架之间存在一种层次关系。一般称前者为上层框架（或父框架），后者为下层框架（或子框架）。当然，上层和下层是相对而言的。例如，“大学教师”，虽然是“教师-1”的上层框架，但它却是“教师”框架的下层框架，而“教师”又是“知识分子”的下层框架。

框架之间的这种层次关系对减少信息冗余有重要意义。因为上层框架与下层框架所表示的事物，在逻辑上为种属关系，即一般与特殊的关系。这样凡上层框架所具有的属性，下层框架也一定具有。于是下层框架就可以从上层框架那里“继承”某些槽值或侧面值，所以“特性继承”也就是框架这种知识表示方法的一个重要特征。

进一步考查上例可以看出，由于一个框架的槽值还可以是另一个框架的名，这就把有关框架横向联系了起来。框架间的“父子”关系是框架间的一种纵向联系。于是，某一论域的全体框架便构成一个框架网络或框架系统。另外，还可看到框架的槽值一般是属性值或状态值，但也可以是规则或逻辑式、运算式甚至过程调用等，例如，上面的工龄就是一个运算式子。

2.5.2 框架的BNF描述

下面给出框架的BNF描述：

〈框架〉 ::= 〈框架头〉 〈槽部分〉 [〈约束部分〉]

〈框架头〉 ::= 〈框架名〉 〈框架名的值〉

〈槽部分〉 ::= 〈槽〉, [〈槽〉]

〈约束部分〉 ::= 〈约束〉 〈约束条件〉, [〈约束条件〉]

〈框架名的值〉 ::= 〈符号名〉 | 〈符号名〉 (〈参数〉, [〈参数〉])

〈槽〉 ::= 〈槽名〉 〈槽值〉 | 〈侧面部分〉

〈槽名〉 ::= 〈系统预定义槽名〉 | 〈用户自定义槽名〉

〈槽值〉 ::= 〈静态描述〉 | 〈过程〉 | 〈谓词〉 | 〈框架名的值〉 | 〈空〉

〈侧面部分〉 ::= 〈侧面〉, [〈侧面〉]

〈侧面〉 ::= 〈侧面名〉 〈侧面值〉

〈侧面名〉 ::= 〈系统预定义侧面名〉 | 〈用户自定义侧面名〉

〈侧面值〉 ::= 〈静态描述〉 | 〈过程〉 | 〈谓词〉 | 〈框架名的值〉 | 〈空〉

〈静态描述〉 ::= 〈数值〉 | 〈字符串〉 | 〈布尔值〉 | 〈其他值〉

〈过程〉 ::= 〈动作〉 | 〈动作〉, [〈动作〉]

〈参数〉 ::= 〈符号名〉

对此表示有如下几点说明:

(1) 框架名的值允许带有参数。此时, 当另一个框架调用它时需要提供相应的实在参数。

(2) 当槽值或侧面值是一个过程时, 它既可以是一个明确表示出来的〈动作〉串, 也可以是对主语言的某个过程的调用, 从而可将过程性知识表示出来。

(3) 当槽值或侧面值是谓词时, 其真值由当时谓词中变元的取值确定。

(4) 槽值或侧面值为〈空〉时, 表示该值等待以后填入, 当时还不能确定。

(5) 〈约束条件〉是任选的, 当不指出约束条件时, 表示没有约束。

2.5.3 框架系统中的预定义槽名

在框架系统中, 框架之间的联系实际上是通过在槽中填入相应的框架名来实现的。为了提供一些常用且可公用的槽名, 在框架系统中通常预先定义了一些标准槽名, 称这些槽名为系统预定义槽名。常用的预定义槽名有以下几种:

(1) is_a 槽

is_a 槽用来指出一个具体事物与其抽象概念间的类属关系。其直观含义为“是一个”, 表示一个事物是另外一个事物的特例。设 F 和 G 是两个实体框架, 则“ F is_a G ”的含义为实体集 F 为实体集 G 的一个特例。一般来说, is_a 槽所指出的联系都具有继承性, 即下层框架可以继承上层框架所描述的属性或值。

(2) AKO 槽

AKO 槽用来指出事物间在抽象概念上的类属关系。其直观含义为“是一种 (A kind of)”, 表示一个事物是另外一个事物的一种类型, 例如, 分类问题。用 AKO 作为下层框架的槽名时, 其槽值为上层框架的框架名。它表示该下层框架所描述的事物比其上层框架更具体, 或者说该上层框架所描述的事物比下层框架更一般或更抽象。用 AKO 作为下层框架的槽名时, 说明下层框架对上层框架具有继承性, 即下层框架可以继承其上层框架所描述的属性或值。

(3) subclass 槽

subclass 槽用来指出类与类之间的类属关系。当用它作为某下层框架的槽时, 表示该下层框架是其上层框架的一个子类。

(4) instance 槽

instance 槽用来建立 AKO 槽的逆关系。当用它作为某上层框架的槽时, 可用来指出它的下一层框架都有哪些。由 instance 槽所建立起来的上、下层框架间的联系具有继承性, 即下层框架可以继承上层框架所描述的属性与值。

(5) part_of 槽

part_of 槽用于指出“部分”与“全体”的关系。当用它作为某下层框架的槽时, 它指出该下层框架所描述的事物仅是其上层框架的一部分。例如, 上层框架描述的是“人体”, 而下层框架描述的是“手”。显然, 手仅是人体的一部分。

需要指出的是, `part_of` 槽与前面提到的 4 种槽在本质上是有所区别的。前面 4 种槽描述的都是上、下层框架之间的类属关系, 它们之间具有共同特征, 且具有继承性。`part_of` 槽仅是指出下层框架为上层框架的子结构, 它们之间一般不具有共同特征, 也不具有继承性。

(6) `infer` 槽

`infer` 槽用于指出两个框架所描述事物间的逻辑推理关系, 它可用来表示相应的产生式规则。例如, 有如下知识。

框架名: 〈诊断规则〉

症状 1: 咳嗽

症状 2: 发烧

症状 3: 打喷嚏

`infer`: 〈结论〉

可信度: 0.8

框架名: 〈结论〉

病名: 感冒

用药: 口服感冒清

服法: 一日 3 次, 每次 2 粒

(7) `possible_reason` 槽

`possible_reason` 槽与 `infer` 槽的作用相反, 用来把某个结论与可能的原因联系起来。例如, 在上述“结论”框架中, 可增加一个 `possible_reason` 槽, 其槽值为某个框架的框架名。

(8) `similar` 槽

`similar` 槽用于指出两个框架所描述的事物之间的相似关系。如果两个框架所表示事物的成员之间有足够多的共同特性, 则认为它们是相似的。这种相似关系用 `similar` 槽描述。

(9) `rotation` 槽

框架可以通过完全任意的关系相联。例如, `rotation` 槽用于指出两个框架所描述的事物之间的“旋转”关系。如果两个框架所表示的事物是从两个不同角度对同一实体的观察, 则它们之间可以通过 `rotation` 槽相连。

2.5.4 框架系统的问题求解过程

由框架的形式可以看出, 框架适合表达结构性的知识。所以, 概念、对象等知识最适于用框架表示。其实, 框架的槽就是对象的属性或状态, 槽值就是属性值或状态值。不仅如此, 框架还可以表示行为(动作), 所以有些过程性事件或情节也可用框架网络来表示。这是框架系统的表达能力。

在框架理论中, 框架是知识的基本单位, 把一组有关的框架连接起来便可形成一个框架系统。在框架系统中, 系统的行为由该系统内框架的变化来实现, 系统的推理过程由框架之间的协调来完成。

用框架表示知识的系统中, 问题求解主要是通过匹配与填槽实现的。用框架求解问题的基本过程如下:

(1) 将问题用适当的框架表示出来。

- (2) 与数据库中已有的框架进行匹配。
- (3) 确定可匹配的预选框架, 进一步收集信息。
- (4) 选用适当的评价方法对预选框架进行评价, 决定其是否被接受。

2.5.5 框架系统的程序语言实现

框架系统可以方便地用程序语言进行实现。有一种名为框架表示语言 (frame representation language, FRL) 的程序设计语言, 就是专门基于框架的程序设计语言。用它就可以方便地实现框架知识表示。不过, 用 Prolog 也可方便地实现框架表示。

用 Prolog 实现框架表示, 一般采用含结构或表的谓词来实现。因为框架实际上就是树, 而 Prolog 的结构也是树, 表又是特殊的结构, 它的元素个数和层数都不限定, 可动态变化, 因此, 更适于表示一般的框架。例如, 前面的“教师”框架用 Prolog 可表示如下:

```
frame (name (“教师”),
      kind_of (“〈知识分子〉”),
      work (scope (“教学”, “科研”), default (“教学”))
      sex (“男”, “女”),
      reco_of_f_s (“中师”, “高师”),
      type (“〈小学教师〉”, “〈中学教师〉”, “〈大学教师〉”)).
```

关于框架的通用表示形式, 可参考有关 Prolog 树、表等章节的内容。

2.5.6 框架系统的特点

框架表示方法的主要优点如下:

- ① 有利于期望引导的处理。
- ② 在给定的状况下, 通过设计能决定其本身的可利用性或者提供其他框架。
- ③ 深层次, 结构性和一致性较好, 并且具有继承性。
- ④ 知识组织的方式有利于推理。

其主要缺点如下:

- ① 许多实际情况与原型不符。
- ② 不善于表达过程性知识。
- ③ 属性的不确定性带来知识表示的不确定性。
- ④ 对新的情况、特例及复合对象的描述能力欠缺。

框架表示法通常适用于表述数学概念及相对较窄的专业知识领域。

2.6 脚本表示法

脚本表示法是夏克 (R. C. Schank) 依据他的概念依赖理论提出的一种知识表示方法, 时间约在 1975 年。脚本与框架类似, 由一组槽组成, 用来表示特定领域内一些事件的发生

序列。

2.6.1 概念依赖理论

在人类的各種知識中，常識性知識是數量最大、涉及面最寬、關係最複雜的知識，很難把它們形式化地表示出來交給計算機處理。面對這一難題，夏克提出了概念依賴理論，其基本思想是：把人類生活中各類故事情節的基本概念抽取出來，構成一組原子概念，確定這些原子概念間的相互依賴關係，然後把所有故事情節都用這組原子概念及其依賴關係表示出來。

由於各人的經歷不同，考慮問題的角度和方法不同，因此抽象出來的原子概念也不盡相同，但一些基本要求卻是應該遵守的。例如，原子概念不能有二義性，各原子概念應該互相獨立等。夏克在其研製的 SAM (script applier mechanism) 中對動作一類的概念進行了原子化，抽取了 11 種原子動作，並把它們作為槽來表示一些基本行為。這 11 種原子動作如下。

(1) PROPEL: 表示對某一對象施加外力，例如，推、拉、打等。

(2) GRASP: 表示行為主體控制某一對象，例如，抓起某件東西，扔掉某件東西等。

(3) MOVE: 表示行為主體變換自己身體的某一部位，例如，抬手、蹬腳、站起、坐下等。

(4) ATRANS: 表示某種抽象關係的轉移。例如，當把某物交給另一人時，該物的所有關係就發生了轉移。

(5) PTRANS: 表示某一物理對象物理位置的改變。例如，某人從一處走到另一處，其物理位置發生了變化。

(6) ATTEND: 表示用某個感覺器官獲取信息。例如，用眼睛查看或用耳朵聽某種聲音等。

(7) INGEST: 表示把某物放入體內，例如，吃飯、喝水等。

(8) EXPEL: 表示把某物排出體外，例如，落淚、嘔吐等。

(9) SPEAK: 表示發出聲音，例如，唱歌、喊叫、說話等。

(10) MTRANS: 表示信息的轉移，例如，看电视、窃听、交谈、读报等。

(11) MBUILD: 表示由已有的信息形成新信息。

夏克利用這 11 種原子概念及其依賴關係把生活中的事件編制成腳本，每個腳本代表一類事件，並把事件的典型情節規範化。當接受一個故事時，就找出一個相應的腳本與之匹配，根據事先安排的腳本情節來理解故事。

2.6.2 腳本的结构

腳本表述的是特定範圍內的原型事件的結構，它是框架的一種特殊形式，用一組槽來描述這些事件的發生序列。腳本通常由開場條件、角色、道具、場景和結局等幾部分组成。一個腳本通常由以下幾部分组成。

(1) 進入條件：給出在腳本中所描述事件的前提條件。

(2) 角色：一些用來表示在腳本所描述事件中可能出現的有关人物的槽。

(3) 道具：一些用来表示在脚本所描述事件中可能出现的有关物体的槽。

(4) 场景：用来描述事件发生的真实顺序。一个事件可以由多个场景组成，而每个场景又可以是其他的脚本。

(5) 结局：给出在脚本所描述事件发生以后所产生的结果。

下面以夏克的“餐厅”脚本为例来说明各个部分的组成。

(1) 进入条件：

① 顾客饿了，需要进餐；

② 顾客有足够的钱。

(2) 角色：顾客，服务员，厨师，老板。

(3) 道具：食品，桌子，菜单，钱。

(4) 场景分别如下，

场景 1：进入—— ① 顾客进入餐厅；

② 寻找桌子；

③ 在桌子旁坐下。

场景 2：点菜—— ① 服务员给顾客菜单；

② 顾客点菜；

③ 顾客把菜单还给服务员；

④ 顾客等待服务员送菜。

场景 3：等待—— ① 服务员告诉厨师顾客所点的菜；

② 厨师做菜，顾客等待。

场景 4：吃饭—— ① 厨师把做好的菜给服务员；

② 服务员把菜送给顾客；

③ 顾客吃菜。

场景 5：离开—— ① 服务员拿来账单；

② 顾客付钱给服务员；

③ 顾客离开餐厅。

(5) 结果：

① 顾客吃了饭，不饿了；

② 顾客花了钱；

③ 老板赚了钱；

④ 餐厅食品少了。

2.6.3 脚本的推理

脚本描述的事件是一个因果链。链头是一组开场条件，只有当这些初始条件满足时，该脚本中的事件才能开始；链尾是一组结果，只有当这一组结果满足时，该脚本中的事件才能结束，以后的事件或事件序列才能发生。在这个因果链中，一个事件和其前后事件之间是相互联系的，前面的事件可使当前事件产生，当前事件又可能使后面的事件产生。

一个脚本建立之后，如果已知该脚本适合于所给定的事件，则对一些在脚本中没有明

显提出的事件，可以通过脚本进行预测，对那些在脚本中已明显提到的事件，可通过脚本给出它们之间的联系。一个脚本在使用之前必须先将其激活，根据脚本的重要性，激活脚本有以下两种方法：

(1) 对于不属于事件核心部分的脚本，只需要设置指向该脚本的指针即可，以便当它成为核心时能够启用。例如，对于前面讨论过的餐厅脚本，有以下事件：

“何雨在去展览会的路上经过他喜欢的饭馆。他非常喜欢这次网络信息展览会。”那么应该采用这种方法，设置指向餐厅脚本的指针。

(2) 对于符合核心事件的脚本，则应使用在当前事件中涉及的具体对象和人物去填写脚本槽。剧本的前提、道具、角色和事件等常能起到激活脚本的指示器的作用。

一旦脚本被激活，则可以用它来进行推理。其中，最重要的是利用剧本可以预测没有明确提及的事件的发生。例如，对于以下情节：

“昨晚，何雨到了餐厅，他订了鱼香肉丝、大米。当他要付款时发现没钱了。因为开始下雨了，所以他赶快回家了”。

如果有人问：

“昨晚，何雨吃饭了吗？”

虽然上面没有提到何雨吃没吃饭的问题，但借助于餐厅剧本，可以回答：“他吃了”。这是因为启用了餐厅剧本。情节中的所有事件与剧本中所预测的事件序列相对应，因此可以推出整个事件正常进行时所得出的结果。

但是，一旦一个典型的事件被中断，也就是给定情节中的某个事件与剧本中的事件不能对应时，则剧本便不能预测被中断以后的事件了。例如，有如下情节：

“何雨走进餐厅，他被带到餐桌旁，订了一大盘鱼香肉丝和米饭之后，他坐在那里等了许久。于是，他生气地走了。”

在该情节中，因为要久等，所以何雨走了。这一事件改变了餐厅脚本中所预测的事件序列，因而被中断了。这时就不能推断何雨是否付了账等情节。但是，仍然可以推出他看了菜单，这是因为看菜单事件发生在中断之前。从这个例子还可以看出，利用脚本可以将事情的注意力集中在“因为久等，何雨生气了”这样一些特殊事件的发生上。

2.6.4 脚本表示法的特点

脚本结构与框架结构相比要呆板得多，知识表示的范围也比较窄。因此，脚本无法有效表示生活当中多种多样的知识。但是，对于表达预先构思好的特定知识或顺序性动作及事件，如理解故事情节等，是非常有效的。目前，脚本表示主要在自然语言理解方面获得了一些应用。

2.7 过程表示法

在人工智能的发展史中，关于知识的表示方法曾存在两种不同的观点。一种观点认为知识主要是陈述性的，其表示方法应着重将其静态特性，即事物的属性以及事物间的关系

表示出来，称以这种观点表示知识的方法为陈述式或说明性表示方法；另一种观点认为知识主要是过程性的，其表示方法应将知识及如何使用这些知识的控制性策略均表述为求解问题的过程，称以这种观点表示知识的方法为过程性表示方法或过程表示法。

2.7.1 表示知识的方法

说明性表示方法是一种静态表示知识的方法，其主要特征是把领域内的过程性知识与控制性知识（即问题求解策略）分开。如在前面讨论的产生式系统中，规则库只是用来表示并存储领域内的过程性知识，而把控制性知识隐含在控制系统中，两者是分离的。

过程性表示方法着重于对知识的利用，它把与问题有关的知识以及如何运用这些知识求解问题的控制策略都表述为一个或多个求解问题的过程，每个过程是一段程序，用于完成对一个具体事件或情况的处理。在问题求解过程中，当需要使用某个过程时就调用相应的程序并执行。在以这种方法表示知识的系统中，知识库是一组过程的集合，当需要对知识库进行增、删、改时，则相应地增加、删除及修改有关的过程。

前面所讨论的几种知识表示方法，均属陈述性知识表示，它们所强调的是知识的静态、显式描述，而对于如何使用这些知识，则需要通过控制策略来决定。过程性知识表示则不同，它可将有关某一问题领域的知识，连同如何使用这些知识的方法，均隐式地表示为一个求解问题的过程。

过程性知识表示方法中，过程所给出的是事物的一些客观规律，表达的是如何求解问题，知识的描述形式就是程序，所有信息均隐含在程序之中。过程表示没有固定的表示形式，如何描述知识完全取决于具体问题。下面以过程规则表示形式为例来说明知识的过程表示问题。一般来说，一个过程规则由以下4部分组成。

（1）激发条件。激发条件由推理方向和调用模式两部分组成。其中，推理方向用于指出推理是正向推理（FR）还是逆向推理（BR）。若为正向推理，则只有当综合数据库中的已有事实可以与其“调用模式”匹配时，该过程规则才能被激活。若为逆向推理，则只有当“调用模式”与查询目标或子目标匹配时才能将该过程规则激活。

（2）演绎操作。演绎操作由一系列的子目标构成。当前面的激发条件满足时，将执行这里列出的演绎操作。

（3）状态转换。状态转换操作用来完成对综合数据库的增、删、改操作。

（4）返回。过程规则的最后一个语句是返回语句，用于指出将控制权返回到调用该过程规则的上一级过程规则那里去。

作为例子，下面给出一个关于同学问题的过程表示。设有如下知识：

“如果 x 与 y 是同班同学，且 z 是 x 的老师，则 z 也是 y 的老师”

可用过程规则表示如下：

```
BR(Teacher? z? y)
  GOAL(Classmate? x y)
  GOAL(Teacher z x)
  INSERT(Teacher z y)
  RETURN
```

其中，BR 是逆向推理标志；GOAL 表示求解子目标，即进行过程调用；INSERT 表示对数据库进行插入操作；RETURN 作为结束标志；带“？”的变量表示其值将在该过程中求得。

2.7.2 过程表示的问题求解过程

在用过程规则表示知识的系统中，问题求解的基本过程是：每当有一个新的目标时，就从可以匹配的过程规则中选择一个执行。在该规则的执行过程中可能会产生新的目标，此时就调用相应的过程规则并执行它。反复进行这一过程，直至执行到 RETURN 语句，这时将控制权返回给调用当前过程的上一级过程规则，并按照调用时的相反次序逐级返回。在这一过程中，如果某过程规则运行失败，就另选择一个同层的可匹配的过程规则执行，如果不存在这样的过程规则，则返回失败标志，并将执行的控制权移交给上一级过程规则。

下面仍以上述例子为例来说明采用正向推理的问题求解过程。

设综合数据库中有以下已知事实：

(Classmate 王涛 赵晔)
(Teacher 林海 王涛)

其中，第 1 个事实表示王涛与赵晔是同班同学；第 2 个事实表示林海是王涛的老师。

假设需要求解的问题是：找出两个人 w 及 v ，其中 w 是 v 的老师。该问题可表示如下：

GOAL(Teacher?w?v)

求解该问题的过程如下：

(1) 在过程规则库中找出对于问题 GOAL (Teacher? w ? v)，其激发条件可以满足的过程规则。显然，BR (Teacher? z ? y) 经如下变量代换：

$w / z, v / y$

之后可以匹配，因此选用该过程规则。

(2) 执行该过程规则中的第 1 个语句 GOAL (Classmate? x y)。此时，其中的 y 已被 v 替换。经与已知事实 (Classmate 王涛 赵晔) 匹配，分别求得了变量 x 及 v 的值，即

$x = \text{王涛} \quad v = \text{赵晔}$

(3) 执行该过程规则中的第 2 个语句 GOAL (Teacher z x)。此时， x 的值已经知道， z 已被 w 替换。经与已知事实 (Teacher 林海 王涛) 匹配，求得了变量 w 的值，即

$w = \text{林海}$

(4) 执行该过程规则中的第 3 个语句 INSERT (Teacher z y)，此时， z 与 y 的值均已知，分别是林海和王涛，因此这时插入数据库的事实如下：

(Teacher 林海 赵晔)

这表明“林海也是赵晔的老师”，求得了问题的解。

2.7.3 过程表示的特点

过程表示法的知识有如下优点：

(1) 表示效率高

过程表示法是用程序来表示知识的，而程序能准确地表明先做什么、后做什么以及怎样做，并直接嵌入一些启发式的控制信息。因此，可以避免选择及匹配那些无关的知识，也不需要跟踪那些不必要的路径，从而提高了系统的运行效率。

(2) 控制系统容易实现

由于控制性质已嵌入到程序中，因而控制系统就比较容易设计。

过程表示法的主要缺点是不易修改和添加新知识，而且当对某一过程进行修改时，又可能影响到其他过程，为系统的维护带来不便。

目前的发展趋势是探讨说明性与过程性相结合的知识表示方法，以便在可维护性、可理解性以及运行效率方面寻求一种比较合理的解决方法。

2.7.4 过程性与说明性表示方法的比较

说明性的知识表示方法是对知识和事实的一种静态描述，是对知识的一种显式表达形式。过程性表示方法则是将有关某一问题领域的知识，连同如何使用这些知识的方法，都隐式地表达为一个求解问题的过程。它强调的是知识的动态方面，即如何发现相关的知识并进行推理等。因此过程能够最好地体现知识的行为。

过程表示是将知识包含在若干过程之中。过程是一小段程序，它处理某些特殊事件或特殊状况。每个过程都包含说明客体和事件的知识，以及在说明完好的情况下的运行知识等。过程通常用子程序或模块实现。采用过程表示知识库的特征为：知识库是一组过程集合。知识库的修改是增加、删除子程序，或修改子程序及访问条件。

过程表示方法与说明性表示方法比较，其主要优点是：

- ① 有利于表示启发式知识。
- ② 能实现扩充逻辑推理（如默认推理等）。
- ③ 具有高度模块化的优点。
- ④ 能够通过类比进行推理。

其主要缺点是：

- ① 由于知识隐含在过程之中，因此难于修改和维护。
- ② 固定的控制信息限定了其他可能的方法。过程表示方法本身的适用范围比较窄，但是研究说明性表示与过程性表示相结合的高效、易维护、可理解的知识表示方法却是很有意义的。

2.8 Petri 网表示法

Petri 网的概念是 1962 年由德国学者 Cah Abam Petri 提出的。开始用于构造系统模型及进行动态特性分析，后来逐渐被用于知识表示。

2.8.1 Petri 网的基本概念

Petri 网表示法中，对于不同的应用，网的构成及构成元素的意义均不相同。但有 3 种元素是基本的，它们是位置、转换及标记。这 3 种元素之间的关系如图 2.8 所示。

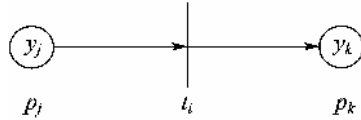


图 2.8 Petri 网

图中 p_j 与 p_k 分别代表第 j 个和第 k 个位置， y_j 与 y_k 分别是这两个位置的标记， t_i 是某一转换。

2.8.2 表示知识的方法

如果用 p_j 与 p_k 分别对应产生式规则的前提 d_j 和结论 d_k ，用 t_i 代表规则强度 μ_i ，则图 2.8 所示的 Petri 网就与如下产生式规则有相同的含义：

IF d_j THEN d_k (CF= μ_i)

对于比较复杂的知识，Petri 网通常用一个 8 元组来表示知识之间的因果关系。具体形式如下：

$$(P, T, D, I, O, f, \alpha, \beta)$$

其中：

P 是位置的有限集，记为 $P = \{p_1, p_2, \text{L}, p_n\}$ ；

T 是转换的有限集，记为 $T = \{t_1, t_2, \text{L}, t_n\}$ ；

D 是命题的有限集，记为 $D = \{d_1, d_2, \text{L}, d_n\}$ ；

I 是输入函数，表示从位置到转换的映射；

O 是输出函数，表示从转换到位置的映射；

f 是相关函数，表示从转换到 0~1 之间一个实数的映射，用来表示规则强度；

α 是相关函数，表示从转换到 0~1 之间一个实数的映射，用来表示位置对应命题的可信度；

β 是相关函数，表示从位置到命题的映射，用于表示位置所对应的命题。

在上面的叙述中，用到了“规则强度”及“可信度”的概念，这是用来表示不确定性知识的。对于知识的不确定性有多种表示方法，“可信度”是其中的一种，它用来指出对知识为真的相信程度，通常用[0, 1]上的一个实数表示，值越大表示相信为真的程度越高。对于一个产生式规则，其可信度称为规则强度。

设有如下产生式规则：

IF d_j THEN d_k (CF= μ_i)

若 d_j 的可信度为 0.8，规则强度 $\mu_i = 0.9$ ，则 Petri 网中各元素的内容分别如下：

$$\begin{aligned}
 P &= \{p_j, p_k\} & T &= \{t_i\} & D &= \{d_j, d_k\} \\
 I(t_i) &= \{p_j\} & O(t_i) &= \{p_k\} & f(t_i) &= \mu_i = 0.9 \\
 \alpha(p_j) &= 0.8 & \beta(p_j) &= d_j & \beta(p_k) &= d_k
 \end{aligned}$$

Petri 网表示法的基本思想就是用不同的位置来代表产生式规则的前提及结论，用转换来表示不同的规则强度，从而实现 Petri 网对产生式规则的规则集的映射。

再如，对于如下产生式规则集

```

r1: IF d1 THEN d2 (CF=0.85)
r2: IF d2 THEN d3 (CF=0.8)
r3: IF d2 THEN d4 (CF=0.8)
r4: IF d4 THEN d5 (CF=0.9)
r5: IF d1 THEN d6 (CF=0.9)
r6: IF d6 THEN d9 (CF=0.93)
r7: IF d1 AND d8 THEN d7 (CF=0.9)
r8: IF d7 THEN d4 (CF=0.9)

```

则可根据此画出对应的 Petri 网图（留给读者完成）。

2.8.3 Petri 网表示法的特点

Petri 网表示法的主要特点如下：

- (1) 便于描述系统状态的变化及对系统特性的分析。
- (2) 可在不同层次上变换描述，不必注意细节及相应的物理表示。
- (3) 适用于表述不确定性知识。

2.9 面向对象表示法

面向对象是 20 世纪 90 年代软件的核心技术之一，并已在计算机学科的众多领域中得到了成功应用。目前，面向对象技术已经取得了长足的进步，其研究已经涉足计算机软、硬件的多个领域，如面向对象程序设计方法学、面向对象数据库、面向对象操作系统、面向对象软件开发环境、面向对象硬件支持等。

在人工智能领域，人们已经把面向对象的思想、方法用于智能系统的设计与开发，并在知识表示、知识库组成与管理、专家系统设计等方面取得了较大进展。

2.9.1 面向对象的基本概念

对象、类、封装、继承是面向对象技术中的基本概念，对于理解面向对象的思想及方法有十分重要的作用。对象是由一组数据和与该组数据相关的操作构成的实体。在面向对象表示中，类和类继承是重要概念。类由一组变量和一组操作组成，它描述了一组具有相

同属性和操作的对象。每个对象都属于某一个类，每个对象都可由相关的类生成，类生成的过程就是例化。一个类拥有另一个类的全部变量和操作，这种拥有就是继承，继承是面向对象表示法的主要推理形式。

1. 对象

“对象”是人们日常生活中经常用到的一个词汇，但到目前为止还没有一个统一的定义。从广义上讲，所谓“对象”是指客观世界中的任何事物，即任何事物都可以在一定前提下成为被认识的对象，它既可以是一个具体的简单事物，也可以是由多个简单事物组合而成的复杂事物。从这个意义上讲，整个世界也可被认为是一个最复杂的对象。

从问题求解的角度来讲，对象是与问题领域有关的客观事物。由于客观事物都具有其自然属性及行为，因此当把与问题有关的属性及行为抽取出来加以研究时，相应客观事物就在这些属性及行为的背景下成为所关心的对象。

按照哲学的观点，对象有两重性，即对象的静态描述和动态描述。其中，静态描述表示对象的类别属性；动态描述表示对象的行为特性。它们之间相互影响，相互依存。在面向对象系统中，对象是系统中的基本单位，它的静态描述可表示为一个4元组：

对象 $::= (\text{ID}, \text{DT}, \text{OP}, \text{FC})$

其中，ID 是对象的名字；DT 是对象的数据；OP 是对象的操作；FC 是对象的外部接口。

从对象的实现机制来讲，对象是一台自动机，它有一个名字，有一组数据和一组操作，不同对象间的相互作用通过互传消息实现。其中，数据表示对象的状态。操作分为两类，一类用于对数据进行操作，改变对象的状态，另一类用于产生输出结果。对一个对象来说，其他对象的操作不能直接操纵该对象私有的数据，只有对象私有的操作可以操纵它，即对象的状态只能由它私有的操作改变它。可见，对象是把数据和操作该数据的代码封装在一起的实体。

由对象的自动机表示可以看出，对象是一个具有局部状态和一个操作集合的实体，而且数据与操作是不可分的。

2. 类

类是一种对象类型，它描述同一类型对象的共同特征。这种特征包含操作特征和存储特征。类具有继承性，一个类可以是某一类的子类，子类可以继承父类的所有特征。类的每个对象都可作为该类的一个实例。

类可用一个5元组形式地描述如下：

类 $::= (\text{ID}, \text{INH}, \text{DT}, \text{OI}, \text{IF})$

其中，ID 和 DT 与对象的含义相似；INH 是类的继承描述；OI 是操作集；IF 是外部接口。

3. 消息与方法

消息传递是对象之间进行通信的惟一手段，一个对象可以通过传递消息与别的对象建立联系。消息是对象之间相互请求或相互协作的途径，是要求某个对象执行其中某个功能操作的规格说明。消息的功能是请求对象执行某种操作。方法是对对象实施各种操作的描述，即消息的具体实现。

2.9.2 面向对象的基本特征

面向对象的基本特征主要体现在模块性、封装性、继承性、多态性、易维护性等方面。

1. 模块性

在面向对象系统中，对象是一个可以独立运行的实体，其内部状态不直接受外界的影响，它具有模块化的两个重要特性：抽象和信息隐蔽。模块性是设计良好软件系统的基本属性。

2. 封装性

封装是一种信息隐蔽技术。它是指把一个数据和与这个数据有关的操作集合放在一起，形成一个封装体，外界只需要知道其功能而不必知道实现细节即可使用。对象作为一个封装体，可以把其使用者和设计者分开，从而便于进行软件的开发。

3. 继承性

继承所表达的是一种对象类之间的相交关系，它使得某类对象可以继承另一类对象的特征和能力。继承性是通过类的派生来实现的，如果一个父类派生了一个子类，则子类可以继承其父类的数据和操作。继承性可以减少信息冗余，实现信息共享。

4. 多态性

多态性是指一个名字可以有多种含义。例如，对运算符“+”，它可以做整数的加，也可以做实数的加，甚至还可以做其他数据类型的加。尽管它们使用的运算符相同，但所对应的代码却不同，究竟使用哪些代码，由运算时的入口参数的数据类型来确定。

5. 易维护性

由于对象实现了抽象和封装，使得一个对象可能出现的错误仅限于自身，而不易向外传播，这就便于系统的维护。同时，利用对象的继承性，还可以很方便地进行渐增型程序设计。

2.9.3 面向对象的知识表示

面向对象的封装性体现了对象之间的横向联系，面向对象的继承性则体现了对象之间的纵向联系。这两个特性的存在，使得面向对象系统的不同类之间形成了一种类的层次结构。这种类的层次结构支持分类知识的表示和演绎推理方式。

面向对象的知识表示方法类似框架表示法，知识以类为单位按照一定的层次结构进行组织，不同类之间的联系可通过链来实现。以 C++ 为例，类的一般形式如下：

```
class <类名>[ :<父类名>]
```

```
{  
    private:  
        <私有成员>  
    public:  
        <公有成员>  
}
```

其中，**class** 是类说明的关键字；〈类名〉是类的名字，它是该类在系统中的惟一标识；〈父类名〉是可选的，当该类有父类时，用它来指出其父类的名字；**private** 是私有段的标识关键字，它也是可选的，若私有段处于类说明的第 1 部分，则此关键字可以省略；〈私有成员〉是只有该类本身才能访问的成员；**public** 是公有段的标识关键字；〈公有成员〉是允许其他类访问的成员，它提供了类的外部界面。类中的成员，无论是私有成员还是公有成员，都可以包括数据成员和成员函数两种类型。

原则上讲，前面所讨论的各种知识表示方法都可以用面向对象的方法来描述。但对不同的知识表示方法，其描述方式会有所差别。

面向对象表示法对类的一般描述如下：

```
Class <类名> [:<超类名>]  
    [<类变量名表>]  
    Structure  
        <对象的静态结构描述>  
    Method  
        <关于对象的操作定义>  
    Restraint  
        <限制条件>  
End
```

其中，**Class** 是类描述开始符；〈类名〉是该类的名字，它是系统中该类的惟一标识；〈超类名〉是任选的，当该类有父类时，用它指出父类的名字；〈类变量名表〉是一组变量名构成的序列，该类中所有对象都共享这些变量，对该类对象来说它们是全局变量，当把这些变量实例化为一组具体的值时，就得到了该类中的一个具体对象，即一个实例；**Structure** 后面的〈对象的静态结构描述〉用于描述该类对象的构成方式；**Method** 后面的〈关于对象的操作定义〉用于定义对类元素施行的各种操作，它既可以是一组规则也可以是为实现相应操作所需执行的一段程序，在 C++ 中则为成员函数调用；**Restraint** 后面的〈限制条件〉指出该类元素所应满足的限制条件，可用包含类变量的谓词构成，当它不出现时，表示没有限制。

2.9.4 面向对象表示方法的特点

正如上面的描述所示，在面向对象方法中，类、子类、实例构成了一个层次结构，而且子类可以继承父类的数据及操作。这种层次结构及继承机制直接支持了分类知识的表示，而且其表示方法与框架表示法有许多相似之处，知识可以按类以一定层次形式进行组织，类之间通过链实现联系。面向对象表示法的主要特点表现为继承性，灵活，易于维护，

可重用性好等。

2.10 状态空间表示法

状态空间表示法是基于解答空间的问题表示和求解方法。它通过在某个可能的解空间内寻找可行解来求解问题，它是以状态和运算符为基础来表示和求解问题的。

状态是为描述某类不同事物间的差别而引入的一组最少变量 q_0, q_1, \dots, q_n 的有序集合，其矢量形式为 $Q = [q_0, q_1, \dots, q_n]^T$ 。其中，每个元素 $q_i (i = 0, 1, \dots, n)$ 为集合的分量，称为状态变量。给定每个分量的一组值就得到具体的状态，表示为 $Q_k = [q_{0k}, q_{1k}, \dots, q_{nk}]^T$ 。运算符是使问题从一种状态变化为另一种状态的手段，它可以是走步、过程、规则、数学算子、运算符号或逻辑符号等。

问题的状态空间则是一个表示该问题全部可能状态及其关系的图，它包含 3 种说明的集合，即所有可能的问题初始状态集合 S 、操作符集合 F 以及目标状态集合 G 。因此，可把状态空间记为三元状态 (S, F, G) 。

状态空间表示法就是依据状态图示法作为求解问题的主要基础，它是一个通过搜索某个状态空间以求得运算符序列的一个解答过程。具体的求解过程如下。

(1) 准备：描述初始状态，确定操作符集合，描述目标状态特性。

(2) 求解：从初始状态出发，每次加一个操作符产生新的状态描述，并递增地建立起操作符的相应实验序列，直至达到目标状态为止。

(3) 检验：查看求解过程中产生的新状态，确定其是否与目标状态描述相匹配。相对于最优化问题还要寻求某一准则下的最优化路径。

状态空间表示法的特点：思路简单，清晰明确，操作简便。但是由于它需要扩展结点，当解决复杂问题时就容易产生“组合爆炸”。因此，这种方法更适用于求解简单问题。

2.11 问题归约表示法

问题归约法的基本思想是从目标出发进行逆向推理，通过一系列变换把初始问题变换为子问题集合和子—子问题集合，直至最后归约为一个平凡的本原问题集合。这些本原问题的解可以直接得到，从而解决了初始问题。

问题归约方法应用归约算符把一个问题描述变换为一个归约或后继问题描述的集合。变换所得所有后继问题的解就意味着父辈问题的一个解。所有问题归约的目的是最终产生具有明显解答的本原问题。这些问题可能是能够由状态空间搜索中走动一步来解决的问题，或者可能是别的具有已知解答的更复杂的问题。本原问题除了对终止搜索过程起着明显的作用外，有时还被用来限制归约过程中产生后继问题的替换集合。当一个或多个后继问题属于某个本原问题的指定子集时，就出现这种限制。问题描述可以有多种数据结构形式，如列表、树、字符串、矢量，数组等。采用问题归约表示可由下列 3 部分组成：一个初始问题描述；一套把问题变换为子问题的操作符；一套本原问题描述。

与或图可以有效说明问题归约法的求解途径。通过与或图，把某个单一问题归约为具体应用于某个问题的描述，依次产生出一个中间或结点及其与结点后裔（例外的情况是当子问题集合只含有单项时，在这种情况下，只产生或结点）。与或图中每个结点都代表一个明显的问题或问题集合，其起始结点对应于初始问题描述，终叶结点则对应于本原问题的描述。一个问题求解过程就是由生成与或图的足够部分，并证明起始结点有解而得以完成的。

问题归约方法可以应用状态、算符和目标这些表示法来描述问题，这并不意味着问题归约法和状态空间法是一样的。实际上，递增状态空间搜索应用某个问题归约的普通形式，而且可以把问题归约法看作比状态空间法更通用的问题求解方法。问题归约法能够比状态空间法更有效地表示问题。状态空间法是问题归约法的一种特例。在问题归约法的与或图中，包含有与结点和或结点，而在状态空间法中只含有或结点。

本章小结

知识表示是人工智能研究领域不可忽视的重要研究方向。本章介绍了知识及其表示。

在引入知识相关概念的基础上，对现有的多种知识表示方法：一阶谓词逻辑表示法、产生式表示法、语义网络表示法、框架表示法、脚本表示法、过程表示法、Petri网表示法、面向对象表示法、状态空间表示法、问题归约表示法等进行介绍。分别从它们的基本思想、工作流程、主要特点及相互比较等方面一一进行了分析。

人们在求解问题时总是希望寻求最为便捷、高效的解决途径。对于智能系统而言，知识表示的能力直接关系着系统的运行效率，因此选择适合、高效的知识表示方法，对于求解复杂的智能系统显得尤为重要。由于各种知识表示方法各具特点、各有优劣，并有其适用领域，因而在解决具体问题时，应当把握问题的要旨，综合考虑，选取适当的表示方法。

习题 2

1. 什么是数据、信息和知识？三者之间的关系是什么？
2. 什么是知识表示？目前有哪些常用的知识表示方法？
3. 用状态空间法表示推销员旅行问题。
4. 一阶谓词逻辑表示法具有哪些特点？适用于哪些类型的知识？
5. 用问题归约法和谓词逻辑表示法分别表示梵塔问题的求解过程。
6. 什么是框架？框架表示的一般形式是什么？写出一个“学生框架”的描述。
7. 什么是过程表示法？它具有哪些特点？比较说明性表示方法与过程性表示法。
8. 设有3个传教士和3个野人准备过河。但是只有一条船，该船每次只能负载两人。在岸上的野人人数不得超过传教士的人数，否则传教士就会被野人吃掉。给出所有人都安全渡河的方案，并用适当的知识表示方法将其表示出来。
9. 什么是产生式？产生式与谓词蕴含式有何异同？

10. 什么是产生式系统？它由哪几部分组成？简述产生式系统求解问题的一般步骤。
11. 用产生式表示法表示八数码问题的求解操作。
12. 什么是语义网络？它与其他知识表示方法相比有哪些特点？
13. 用语义网络表示下列命题：
 - (1) 鱼生活在水里。
 - (2) 张强是一名计算机专业的大学三年级学生。
 - (3) 一班与二班进行篮球比赛，最后比分是 72 : 68。
14. 简述语义网络与 Prolog 语言之间的对应关系。用 Prolog 语言表示一种植物分类语义网络。
15. 画出如下产生式规则集的 Petri 网，
 $r1: \text{IF } d_1 \text{ THEN } d_2 \text{ (CF=0.8)}$
 $r2: \text{IF } d_1 \text{ AND } d_3 \text{ THEN } d_4 \text{ (CF=0.7)}$
 $r3: \text{IF } d_2 \text{ AND } d_4 \text{ THEN } d_5 \text{ (CF=0.9)}$
16. 如何用面向对象方法表示知识？
17. 如何对知识表示方法进行评价？如何合理选用知识表示方法？

P 是一个命题常量, 它的真值就是该特定命题的真值。当 P 仅是任意命题的位置标志符时, 可以说 P 是一个命题变元, 它没有真值。在数理逻辑中, 命题变元不代表一个确定的命题, 无法确定其真值, 但可以用任意一个给定的命题取代它, 这时就可以给出 P 的真值。也就是说可以给命题变元任意指派真值。在命题逻辑中常把命题变元简称为变元。

定义 3.2 命题联接词: 在数理逻辑中也有类似的严格定义的联结词, 叫做命题联接词。常用的命题联接词有 5 个, 即合取、析取、否定、蕴含和双条件。它们的定义如表 3.1 所示。

表 3.1 命题连接词逻辑真值表

P	Q	$\sim P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

其中:

“ \sim ”或“ \neg ”叫否定(negation)或补, 其作用是否定位于它后面的命题。它与后面的命题 P 一起构成一个与 P 的真值正好相反的复合命题 $\sim P$ (读作“非 P ”)。

“ \wedge ”叫合取(conjunction), 它表示被它连接的两个命题具有“与”关系。

“ \vee ”叫析取(disjunction), 它表示被它连接的两个命题具有“或”关系。

“ \rightarrow ”叫条件或者蕴含(conditional), $P \rightarrow Q$ 表示 P 蕴含 Q (读作“如果 P , 则 Q ”)。其中, P 称为条件的前件, Q 称为条件的后件。

“ \leftrightarrow ”叫双条件(biconditional), 它联结任意两个命题 P 和 Q , 生成一个复合命题 $P \leftrightarrow Q$ (表示“ P 当且仅当 Q ”)。

定义 3.3 合式公式:

(1) 孤立的命题变元或逻辑常量 (T, F) 是一个合式公式 (这类公式叫原子公式);
 (2) 如果 A 是一个合式公式, 则 $\sim A$ 也是一个合式公式;
 (3) 如果 A 和 B 是合式公式, 则 $(A \wedge B)$ 、 $(A \vee B)$ 、 $(A \rightarrow B)$ 和 $(A \leftrightarrow B)$ 都是合式公式;

(4) 当且仅当经过有限次使用规律 (1)、(2) 和 (3) 得到的由命题变元、联结词符号和圆括号所组成的字符串, 才是合式公式。

为了减少括号的使用次数, 特做如下简化的规定:

(1) 联结词运算的优先级从高到低为 \sim , \wedge , \vee , \rightarrow , \leftrightarrow ;
 (2) 同级联结词中, 先出现的先运算;
 (3) 最外层的括号可以省去, 在上述优先级规定下, 凡省去后不会引起二义性的括号, 均可省去。

定义 3.4 真值指派: 设有一个由 n 个变元 P_1, P_2, \dots, P_n 所组成的谓词公式 A , 则 A 的取值由这 n 个变元惟一确定。如果给 (P_1, P_2, \dots, P_n) 一组确定的值 ($P_i = T$ 或 $F, i=1, 2, \dots, n$), 则 A 有一确定的真值 (T 或 F)。把变元的一组取值叫做公式的一个真值指派。显然, 由 n 个变元组成的公式有 2^n 个不同的真值指派。

真值表：由公式的所有真值指派和对应的公式真值所组成的表叫该公式的真值表。

上述基本联结词的定义就是用真值表给出的，其他任何公式的真值表都可以由基本联结词的真值表导出。例如，公式 $(P \rightarrow Q) \wedge (Q \rightarrow P)$ 的真值表可构造如表 3.2 所示。

表 3.2 公式 $(P \rightarrow Q) \wedge (Q \rightarrow P)$ 逻辑真值表

P	Q	$P \rightarrow Q$	$Q \rightarrow P$	$(P \rightarrow Q) \wedge (Q \rightarrow P)$
F	F	T	T	T
F	T	T	F	F
T	F	F	T	F
T	T	T	T	T

给定一个公式，如果对于所有的真值指派，它的真值都是 T，则称该公式为永真式（或重言式）；如果对于所有的真值指派，它的真值都是 F，则称该公式为永假式（或不可满足的）。除了这两种极端情况外，一般的命题公式的真值有 T 有 F。称非永假的公式为可满足的公式。

定义 3.5 等价 (equivalence)：设 A, B 都是命题公式， $P_i(i=1,2,\cdots,n)$ 是出现在 A 和 B 中的所有命题变元。如果对于这 n 个变元的任何一个真值指派集合， A 和 B 都相等，则称公式 A 等价于公式 B ，记作： $A \Leftrightarrow B$ 。

等价又可定义为：“ $A \Leftrightarrow B$ 当且仅当 $A \Leftrightarrow B$ 是一个永真式”。

定义 3.6 永真蕴含：命题公式 A 永真蕴含命题公式 B ，当且仅当 $A \rightarrow B$ 是一个永真式，记作： $A \Rightarrow B$ ，读作“ A 永真蕴含 B ”，简称“ A 蕴含 B ”。

3.1.2 命题定律

利用真值表，可以证明一批常用的蕴含式和等价式，它们统称为命题定律。

1. 蕴含式

命题定律的蕴含式如下：

- $I_1 \quad P \wedge Q \Rightarrow P$
- $I_2 \quad P \wedge Q \Rightarrow Q$
- $I_3 \quad P \Rightarrow P \wedge Q$
- $I_4 \quad Q \Rightarrow P \wedge Q$
- $I_5 \quad \sim P \Rightarrow P \rightarrow Q$
- $I_6 \quad Q \Rightarrow P \rightarrow Q$
- $I_7 \quad \sim (P \rightarrow Q) \Rightarrow P$
- $I_8 \quad \sim (P \rightarrow Q) \Rightarrow \sim Q$
- $I_9 \quad P, Q \Rightarrow P \wedge Q$
- $I_{10} \quad \sim P, P \vee Q \Rightarrow Q$
- $I_{11} \quad P, P \rightarrow Q \Rightarrow Q$
- $I_{12} \quad \sim Q, P \rightarrow Q \Rightarrow \sim P$

I_{13}	$P \rightarrow Q, Q \rightarrow R \Rightarrow P \rightarrow R$
I_{14}	$P \vee Q, P \rightarrow R, Q \rightarrow R \Rightarrow R$
I_{15}	$(P \rightarrow Q) \Rightarrow (R \vee P \rightarrow R \vee Q)$
I_{16}	$(P \rightarrow Q) \Rightarrow (R \wedge P \rightarrow R \wedge Q)$

2. 等价式

命题定律的等价式可分为以下 7 组。

第 1 组：交换律

$$E_1 \quad P \vee Q \Leftrightarrow Q \vee P$$

$$E_2 \quad P \wedge Q \Leftrightarrow Q \wedge P$$

$$E_3 \quad P \rightarrow Q \Leftrightarrow Q \rightarrow P$$

第 2 组：结合律

$$E_4 \quad (P \vee Q) \vee R \Leftrightarrow P \vee (Q \vee R)$$

$$E_5 \quad (P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R)$$

$$E_6 \quad (P \rightarrow Q) \rightarrow R \Leftrightarrow P \rightarrow (Q \rightarrow R)$$

第 3 组：分配律

$$E_7 \quad P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$$

$$E_8 \quad P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$$

$$E_9 \quad P \rightarrow (Q \rightarrow R) \Leftrightarrow (P \rightarrow Q) \rightarrow (P \rightarrow R)$$

第 4 组：否定律

$$E_{10} \quad \sim \sim P \Leftrightarrow P$$

$$E_{11} \quad \sim (P \wedge Q) \Leftrightarrow \sim P \vee \sim Q$$

$$E_{12} \quad \sim (P \vee Q) \Leftrightarrow \sim P \wedge \sim Q$$

$$E_{13} \quad \sim (P \rightarrow Q) \Leftrightarrow P \wedge \sim Q$$

$$E_{14} \quad \sim (P \rightarrow Q) \Leftrightarrow \sim P \rightarrow Q \Leftrightarrow P \wedge \sim Q$$

$$E_{15} \quad \sim P \rightarrow \sim Q \Leftrightarrow Q \wedge P$$

$$E_{16} \quad \sim P \wedge \sim Q \Leftrightarrow \sim (P \vee Q)$$

第 5 组：吸收律

$$E_{17} \quad P \wedge P \Leftrightarrow P$$

$$E_{18} \quad P \vee P \Leftrightarrow P$$

$$E_{19} \quad (P \wedge Q) \vee P \Leftrightarrow P$$

$$E_{20} \quad (P \vee Q) \wedge P \Leftrightarrow P$$

$$E_{21} \quad P \rightarrow \sim P \Leftrightarrow \sim P$$

$$E_{22} \quad \sim P \rightarrow P \Leftrightarrow P$$

$$E_{23} \quad (P \rightarrow Q) \rightarrow P \Leftrightarrow P$$

$$E_{24} \quad P \rightarrow (P \rightarrow Q) \Leftrightarrow P \rightarrow Q$$

$$E_{25} \quad (P \rightarrow Q) \rightarrow P \Leftrightarrow P$$

第 6 组：常值与变元

E_{26}	$T \wedge P \Leftrightarrow P; F \vee P \Leftrightarrow P$
E_{27}	$F \wedge P \Leftrightarrow F; T \vee P \Leftrightarrow T$
E_{28}	$P \wedge \sim P \Leftrightarrow F$
E_{29}	$P \vee \sim P \Leftrightarrow T$
E_{30}	$T \rightarrow P \Leftrightarrow P; P \rightarrow F \Leftrightarrow \sim P$
E_{31}	$F \rightarrow P \Leftrightarrow T; P \rightarrow T \Leftrightarrow T$
E_{32}	$P \rightarrow P \Leftrightarrow T$
E_{33}	$Q \rightarrow (P \rightarrow Q) \Leftrightarrow T$
E_{34}	$T \quad P \Leftrightarrow P; P \quad P \Leftrightarrow T$
E_{35}	$F \quad P \Leftrightarrow \sim P; P \quad \sim P \Leftrightarrow F$

第7组：连接词化归律

E_{36}	$P \wedge Q \Leftrightarrow \sim(\sim P \vee \sim Q)$
E_{37}	$P \vee Q \Leftrightarrow \sim(\sim P \wedge \sim Q)$
E_{38}	$P \rightarrow Q \Leftrightarrow \sim P \vee Q$
E_{39}	$P \quad Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$
E_{40}	$P \quad Q \Leftrightarrow (P \wedge Q) \vee (\sim P \wedge \sim Q)$

3.1.3 范式

具有相同真值表的公式可以有无穷多个，但它们都是等价的。为了研究方便，需要找到它们的标准形式，即范式。

1. 析取范式

文字：原子公式及其否定叫文字。

基本积：仅由文字构成的合取式叫基本积。

基本和：仅由文字构成的析取式叫基本和。

析取范式：由基本积的析取构成的命题公式叫析取范式。例如，

$$\sim P \vee (P \wedge Q) \vee (P \wedge \sim R) \vee (\sim P \wedge \sim Q)$$

其中的每个基本积叫一个析取项。

任意给定一个命题公式都可以化成与之等价的析取范式。

求公式 A 的析取范式的步骤如下：

- (1) 利用等价式 E_{38} 和 E_{39} （或 E_{40} ）消去公式中的 \rightarrow 和 \leftrightarrow 联结词；
- (2) 利用 E_{11} 和 E_{12} 将联结词 \sim 深入到变元，并用双重否定律 E_{10} 化简到变元前最多只有一个 \sim ；
- (3) 利用分配律 E_7 和 E_8 将公式最后化为析取范式。

2. 合取范式

由基本和的合取构成的命题公式叫合取范式。例如，

$$P \wedge (\sim P \vee Q) \wedge (P \vee \sim Q) \wedge (\sim P \vee \sim R)$$

其中的每个基本和叫合取项。

任意给定一个命题公式 A 都可以化成与之等价的合取范式。

求公式 A 的合取范式的步骤与析取范式类同。

析取范式和合取范式可以解决公式表达的标准化问题，但它们都不惟一，同一公式可以写出许多与之等价的析取范式和合取范式。主范式可以解决公式表达的惟一性问题。

3. 主析取范式

如果析取范式中的每个变元或它的否定形式在每个析取项中必出现一次且仅出现一次，则该析取范式叫主析取范式。

求一个命题公式的主析取范式的方法如下：

- (1) 将给定公式化为析取范式；
- (2) 除去析取范式中所有的永假的析取项（即含有 $P \wedge \sim P$ 的析取项）；
- (3) 若析取项中有同一变元多次出现，则用 $P \wedge P \Leftrightarrow P$ 化简成只出现一次；
- (4) 若给定公式中的变元 Q 或 $\sim Q$ 未出现在某析取项中，则用等价式 $P \Leftrightarrow P \wedge (Q \vee \sim Q) \Leftrightarrow (P \wedge Q) \vee (P \wedge \sim Q)$ 引入 Q 或 $\sim Q$ ，然后除去相同的析取项。

例 1：求公式 $(P \wedge (P \rightarrow Q)) \vee Q$ 的主析取范式。

$$\begin{aligned} (P \wedge (P \rightarrow Q)) \vee Q &\Leftrightarrow (P \wedge (\sim P \vee Q)) \vee Q \\ &\Leftrightarrow (P \wedge \sim P) \vee (P \wedge Q) \vee Q \\ &\Leftrightarrow (P \wedge Q) \vee Q \\ &\Leftrightarrow (P \wedge Q) \vee (P \wedge Q) \vee (\sim P \wedge Q) \\ &\Leftrightarrow (P \wedge Q) \vee (\sim P \wedge Q) \end{aligned}$$

$(P \wedge Q) \vee (\sim P \wedge Q)$ 就是它的主析取范式。

求给定公式的主析取范式的方法有两种，除了上述推导法外，还有一种真值表法，这里不再详细叙述。

4. 主合取范式

如果合取范式中的每个变元或它的否定形式在每个合取项中必出现一次且仅出现一次，则该合取范式叫主合取范式。

求一个命题公式的主合取范式的方法与主析取范式类似。

例 2：求公式 $(P \wedge (P \rightarrow Q)) \vee Q$ 的主合取范式。

$$\begin{aligned} (P \wedge (P \rightarrow Q)) \vee Q &\Leftrightarrow (P \wedge (\sim P \vee Q)) \vee Q \\ &\Leftrightarrow (P \vee Q) \wedge (\sim P \vee Q) \vee Q \\ &\Leftrightarrow (P \vee Q) \wedge (\sim P \vee Q) \end{aligned}$$

$(P \vee Q) \wedge (\sim P \vee Q)$ 就是它的主合取范式。

与求主析取范式一样，除了推导法外，还有真值表法，具体内容略。

3.1.4 命题逻辑的推论规则

在数理逻辑中，关心的仅是论证的有效性，即从给定的前提出发，如何推导出正确的结论。至于前提本身是否正确，则不在研究之列。推论规则是保证推理有效性的一组规则，常用的如下：

(1) 代换规则

用任一公式 A 全部代换永真式 B 中的某个变元 P_i 的所有出现，形成的新公式 B' (叫 B 的代换实例) 仍然是永真式，即 $B \Rightarrow B'$ 。

(2) 取代规则

若 A' 是公式 A 中的子公式且 $A' \Leftrightarrow B'$ ，用 B' 取代 A 中的 A' 的一处或多处出现，所得到的新公式为 B ，则有 $A \Leftrightarrow B$ 。

(3) 分离规则

若公式 A 和 $A \rightarrow B$ 均为永真式，则 B 必为永真式。

为了简化推导过程的书写形式，常用一个公式序列来表示，为此引入另外 3 个规则。

(4) P 规则

在推导的任何步骤上都可以引入前提。

(5) T 规则

在推导时，如果前面步骤中有一个或多个公式永真蕴含公式 S ，则可以把 S 引入推导过程之中。

(6) CP 规则

如果能从 R 和前提集合中推导出 S ，则可从前提集合推导出 $R \rightarrow S$ 。

(7) 反证法

$P \Rightarrow Q$ ，当且仅当 $P \wedge \sim Q \Leftrightarrow F$ 。

例 3: 证明 $R \rightarrow S$ 可由前提 $P \rightarrow (Q \rightarrow S)$ ， $\sim R \vee P$ 和 Q 推导出来。

解：如果将 R 作为假设前提，使它和原前提一起推出 S ，则本题得证。

- | | |
|-------------------------------------|----------------------|
| ① $\sim R \vee P$ | (P) |
| ② R | (P, 假设前提) |
| ③ P | (T, ①, ②, I_{10}) |
| ④ $P \rightarrow (Q \rightarrow S)$ | (P) |
| ⑤ $Q \rightarrow S$ | (T, ③, ④, I_{11}) |
| ⑥ Q | (P) |
| ⑦ S | (T, ⑤, ⑥, I_{11}) |
| ⑧ $R \rightarrow S$ | (CP) |

3.1.5 命题逻辑的局限性

命题逻辑的这种表示法有较大的局限性，它无法把它所描述的客观事物的结构及逻辑特征反映出来，也不能把不同事物间的共同特征表述出来。

例如，对于“老李是小李的父亲”这一命题，若用英文字母表示，例如，用字母 F ，则无论如何也看不出老李与小李的父子关系。

又如，假设要表示下列句子叙述的明显事实：“李明是工人”，用命题逻辑表示时可写为 $LIWORKER$ 。

如果要表示“王华也是工人”，就必须写为 $WANGWORKER$ 。

这是一些完全独立的格式，无法从中得出两者的共同特征，也就无法把两者的共同特征形式地表示出来。如果把这些事实表示如下：

$WORKER(LI)$

$WORKER(WANG)$

那么这就要好得多，因为现在这种表示结构反映出知识本身的结构。如果试图用命题逻辑表示句子：“所有的人都要学习”，那么将面临更大的困难，因为如果现在不对问题进行量化，就必须一个一个地写出已经知道的人都需要学习的独立命题。

这样就不得不应用谓词逻辑作为一种知识表示方法，因为谓词逻辑允许人们表达那些无法用命题逻辑相应地加以表达的事情。正是由于这些原因，谓词逻辑才在命题逻辑的基础上发展起来了。

3.2 一阶谓词逻辑

谓词逻辑是在命题逻辑基础上发展起来的，命题逻辑可看作是谓词逻辑的一种特殊形式。本节主要讨论谓词逻辑的主要概念及有关定理。

3.2.1 谓词

在谓词逻辑中，命题是用谓词表示的，一个谓词可分为谓词名与个体这两个部分。个体表示某个独立存在的事物或者某个抽象的概念；谓词名用于刻画个体的性质、状态或个体间的关系。例如，对于“老张是教师”这个命题，用谓词可表示为 $Teacher(Zhang)$ 。其中， $Teacher$ 是谓词名， $Zhang$ 是个体， $Teacher$ 刻画了 $Zhang$ 的职业是教师这一特征。又如，“ $5>3$ ”这个不等式可用谓词表示为 $Greater(5, 3)$ ，这里 $Greater$ 刻画了 5 与 3 之间的“大于”关系。

谓词的一般形式如下：

$P(x_1, x_2, \dots, x_n)$

其中， P 是谓词名， x_1, x_2, \dots, x_n 是个体。谓词名通常用大写的英文字母表示，个体通常用小写的英文字母表示。

在谓词中，个体可以是常量，也可以是变元，还可以是一个函数。例如，对于“ $x<5$ ”，可表示为 $Less(x, 5)$ ，其中， x 是变元。又如，对于“小王的父亲是教师”，可表示为 $Teacher(father(Wang))$ ，其中 $father(Wang)$ 是一个函数。

在用谓词表示客观事物时，谓词的语义是由使用者根据需要人为定义的。例如，对于谓词 $S(x)$ ，既可以定义它表示“ x 是一个学生”，也可以定义它表示“ x 是一只船”或者别

的什么。

当谓词中的变元都用特定的个体取代时，谓词就具有一个确定的真值：T 或 F。

谓词中包含的个体数目称为谓词的元数。例如， $P(x)$ 是一元谓词， $P(x, y)$ 是二元谓词， $P(x_1, x_2, \dots, x_n)$ 是 n 元谓词。

在谓词 $P(x_1, x_2, \dots, x_n)$ 中，若 $x_i (i = 1, \dots, n)$ 都是个体常量、变元或函数，称它为一阶谓词。如果某个 x_i 本身又是一个一阶谓词，则称它为二阶谓词。余者类推。今后用到的都是一阶谓词。

个体变元的取值范围称为个体域。个体域可以是有限的，也可以是无限的。例如，若用 $I(x)$ 表示“ x 是整数”，则个体域是所有整数，它是无限的。

谓词与函数表面上很相似，容易混淆，其实这是两个完全不同的概念。谓词的真值是“真”或“假”，而函数的值是个体域中的某个个体，函数无真假可言，它只是在个体域中从一个个体到另一个个体的映射。

个体常量、个体变元和函数统称为“项”。

3.2.2 量词

考查两个谓词

$$P(x): x^2-1=(x+1)(x-1)$$

$$Q(x): x+3=1$$

它们都是以有理数为个体域。显然，对于个体域中的所有个体， $P(x)$ 均为 T，然而只有 $x=-2$ 时， $Q(x)$ 才为 T。

如何刻画谓词与个体之间的这两种不同的关系呢？为此引入了一个新的概念——量词。量词有两种：全称量词 ($\forall x$) 和存在量词 ($\exists x$)。

全称量词 ($\forall x$) 读作“对于所有的 x ”，它表示“对个体域中的所有(或任一个)个体 x ”；存在量词 ($\exists x$) 读作“存在 x ”，它表示“在个体域中存在个体 x ”。

符号“ $(\forall x)P(x)$ ”表示命题：“对于个体域中所有的个体 x ，谓词 $P(x)$ 均为 T”。谓词 $P(x)$ 称为 ($\forall x$) 的辖域或作用范围。

符号“ $(\exists x)Q(x)$ ”表示命题：“在个体域中存在某些个体 x ，使谓词 $Q(x)$ 为 T”。谓词 $Q(x)$ 称为 ($\exists x$) 的辖域或存在范围。

当一个一元谓词常量命名式的个体域确定后，经某个量词的作用（叫量化），它将被转化为一个命题，可以确定其真值。例如，对上述谓词 $P(x)$ 和 $Q(x)$ 来说，命题 $(\forall x)P(x)$ ， $(\exists x)Q(x)$ 和 $(\exists x)P(x)$ 的真值都为 T，而命题 $(\forall x)Q(x)$ 的真值为 F。

这就是说，将谓词转化成命题的方法有两种：一是将谓词中的个体变元全部换成确定的个体；二是使谓词量化。

注意：

① 量词本身并不是一个独立的逻辑概念，可以用 \wedge ， \vee 联结词代替。设某个体域是有限集合 S ：

$$S=\{a_1, a_2, \dots, a_n\}$$

由量词的意义可以看出, 对任意谓词 $A(x)$ 有

$$(\forall x)A(x) \Leftrightarrow A(a_1) \wedge A(a_2) \wedge \cdots \wedge A(a_n)$$

$$(\exists x)A(x) \Leftrightarrow A(a_1) \vee A(a_2) \vee \cdots \vee A(a_n)$$

上述关系可以推广到 $n \rightarrow \infty$

② 由量词所确定的命题的真值与个体域有关。如上述命题 $(\exists x)Q(x)$ 的真值, 当个体域是有理数或整数时为 T; 当个体域是自然数时为 F。

有时为了方便起见, 个体域一律用全总个体域, 每个个体变元的真正变化范围则用一个特性谓词来刻画。注意: 对于全程量词, 此特性谓词应作为蕴含的前件; 对于存在量词, 此特性谓词应作为合取式的一项。例如, 用 $R(x)$ 表示 x 为实数, 它刻画了上述 $P(x)$ 和 $Q(x)$ 中的个体变元特性, 则可有下述永真命题成立:

$$(\forall x)(R(x) \rightarrow P(x))$$

$$(\exists x)(R(x) \wedge P(x))$$

$$(\exists x)(R(x) \wedge Q(x))$$

对于二元谓词 $P(x, y)$:

$$(\forall x)(\forall y)P(x, y) \qquad (\forall x)(\exists y)P(x, y)$$

$$(\exists x)(\forall y)P(x, y) \qquad (\exists x)(\exists y)P(x, y)$$

$$(\forall y)(\forall x)P(x, y) \qquad (\exists y)(\forall x)P(x, y)$$

$$(\forall y)(\exists x)P(x, y) \qquad (\exists y)(\exists x)P(x, y)$$

其中, $(\exists x)(\forall y)P(x, y)$ 代表 $(\exists x)((\forall y)P(x, y))$ 。

一般来讲, 量词的先后次序不可交换。例如, x 和 y 的个体域都是所有鞋子的集合, $P(x, y)$ 表示一只鞋 x 可与另一只鞋 y 配对, 则

$$(\exists x)(\forall y)P(x, y)$$

表示“存在一只鞋 x , 它可与任何一只鞋 y 配对”这是不可能的, 是个假命题; 而

$$(\forall y)(\exists x)P(x, y)$$

表示“对任何一只鞋 y , 总存在一只鞋 x 可与它配对”这是可能的, 这是真命题。

3.2.3 谓词逻辑的合式公式

可按下述规则得到谓词逻辑的合式公式:

- (1) 单个谓词是合式公式, 称为原子谓词公式;
- (2) 若 A 是合式公式, 则 $\sim A$ 也是合式公式;
- (3) 若 A 和 B 都是合式公式, 则 $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \leftrightarrow B$ 也都是合式公式;
- (4) 若 A 是合式公式, x 是任一个体变元, 则 $(\forall x)A$ 和 $(\exists x)A$ 也都是合式公式。

在合式公式中, 连接词的优先级别是 \sim 或 \neg , \wedge , \vee , \rightarrow ,

3.2.4 自由变元与约束变元

位于量词后面的单个谓词或者用括弧括起来的合式公式称为量词的辖域, 那么辖域内与量词中同名的变元称为约束变元, 不受约束的变元称为自由变元。例如,

$$(\exists x)(P(x, y) \rightarrow Q(x, y)) \vee R(x, y)$$

其中, $(P(x,y) \rightarrow Q(x,y))$ 是 $(\exists x)$ 的辖域, 辖域内的变元 x 是受 $(\exists x)$ 约束的变元, 而 $R(x,y)$ 中的 x 是自由变元, 公式中的所有 y 都是自由变元。

在谓词公式中, 变元的名字是无关紧要的, 可以把一个名字换成另一个名字。但必须注意, 当对量词辖域内的约束变元更名时, 必须把同名的约束变元都统一改成相同的名字, 且不能与辖域内的自由变元同名; 当对辖域内的自由变元改名时, 不能改成与约束变元相同的名字。例如, 对于公式 $(\forall x)P(x,y)$, 可改名为 $(\forall z)P(z,t)$, 这里把约束变元 x 改成了 z , 把自由变元 y 改成了 t 。

3.2.5 谓词公式的解释

在命题逻辑中, 对命题公式中各个命题变元的一次真值指派称为命题公式的一个解释。一旦解释确定后, 根据各连接词的定义就可求出命题公式的真值(T 或 F)。在谓词逻辑中, 由于公式中可能有个体常量、个体变元以及函数, 因此不能像命题公式那样直接通过真值指派给出解释, 必须首先考虑个体常量和函数在个体域中的取值, 然后才能针对常量与函数的具体取值为谓词分别指派真值。由于存在多种组合情况, 所以一个谓词公式的解释可能有很多个。对于每个解释, 谓词公式都可求出一个真值(T 或 F)。

下面首先给出解释的定义, 然后用例子说明如何构造一个解释以及如何根据解释求出谓词公式的真值。

定义 3.7: 设 D 为谓词公式 P 的个体域, 若对 P 中的个体常量、函数和谓词按如下规定赋值:

- (1) 为每个个体常量指派 D 中的一个元素;
- (2) 为每个 n 元函数指派一个从 D^n 到 D 的映射, 其中

$$D^n = \{(x_1, x_2, \dots, x_n) | x_1, x_2, \dots, x_n \in D\}$$

- (3) 为每个 n 元谓词指派一个从 D^n 到 $\{F, T\}$ 的映射。

则称这些指派为公式 P 在 D 上的一个解释。

例 4: 设个体域 $D = \{1, 2\}$, 求公式 $A = (\forall x)(\exists y)P(x, y)$ 在 D 上的解释, 并指出在每种解释下公式 A 的真值。

解: 在公式 A 中没有包括个体常量和函数, 所以可直接为谓词指派真值, 设

$$P(1,1)=T, P(1,2)=F, P(2,1)=T, P(2,2)=F$$

这就是公式 A 在 D 上的一个解释。在此解释下, 因为 $x=1$ 时, 有 $y=1$ 使 $P(x,y)$ 的真值为 T; $x=2$ 时, 也有 $y=1$ 使 $P(x,y)$ 的真值为 T, 即对于 D 中的所有 x , 都有 $y=1$ 使 $P(x,y)$ 的真值为 T, 所以在此解释下公式 A 的真值为 T。

还可以对公式 A 中的谓词指派另外一组真值, 设

$$P(1,1)=F, P(1,2)=F, P(2,1)=F, P(2,2)=F$$

这是对公式 A 的另一个解释。在此解释下, 对 D 中的所有 x ($x=1$ 与 $x=2$) 不存在一个 y , 使得公式 A 的真值为 T, 所以在此解释下公式 A 的真值为 F。

公式 A 在 D 上共有 16 种解释, 这里不再一一列出, 读者可列出其中的几个, 并求出

公式 A 的真值。

例 5: 设个体域 $D=\{1,2\}$, 求公式 $B=(\forall x)(P(x) \rightarrow Q(f(x),b))$ 在 D 上的某一个解释, 并指出公式 B 在此解释下的真值。

解: 设对个体常量 b , 函数 $f(x)$ 指派的值分别为

$$b=1, f(1)=2, f(2)=1$$

对谓词指派的真值为

$$P(1)=F, P(2)=T, Q(1,1)=T, Q(2,1)=F$$

这里, 由于已指派 $b=1$, 所以 $Q(1,2)$ 与 $Q(2,2)$ 不可能出现, 故没有给它们指派真值。

上述指派就是对公式 B 的一个解释。在此解释下, 由于当 $x=1$ 时, 有

$$P(1)=F, Q(f(1),1)=Q(2,1)=F$$

所以 $P(1) \rightarrow Q(f(1),1)$ 的真值为 T 。 $x=2$ 时

$$P(2)=T, Q(f(2),1)=Q(1,1)=T$$

所以 $P(2) \rightarrow Q(f(2),1)$ 的真值也为 T 。即对个体域 D 中的所有 x 均有

$$(P(x) \rightarrow Q(f(x),b))$$

的真值为 T 。所以公式 B 在此解释下的真值为 T 。

由上面的例子可以看出, 谓词公式的真值都是针对某一个解释而言的, 它可能在某一个解释下的真值为 T , 在另一个解释下的真值为 F 。

3.2.6 含有量词的等价式和蕴含式

显而易见, 命题逻辑中的永真式、等价和蕴含等概念可以推广到谓词逻辑, 命题逻辑中常用的等价式和蕴含式也可以全部推广到谓词逻辑来。一般来说, 只要把原式中的命题公式用谓词公式代替, 且把这种代替贯穿于整个表达式中, 命题逻辑中的永真式就转化成谓词逻辑中的永真式了。

下面介绍谓词逻辑中特有的等价式和蕴含式, 它们是因为量词的引入而产生的, 无论是对有限个体域或是无限个体域, 它们都是正确的。

量词转换律 令 $\sim(\forall x)A(x)$ 表示对整个被量化的命题 $(\forall x)A(x)$ 的否定, 而不是对 $(\forall x)$ 的否定, 于是有

$$\begin{aligned} \sim(\forall x)A(x) &\Leftrightarrow \sim(A(a_1) \wedge A(a_2) \wedge \cdots \wedge A(a_n)) \\ &\Leftrightarrow \sim A(a_1) \vee \sim A(a_2) \vee \cdots \vee \sim A(a_n) \Leftrightarrow (\exists x) \sim A(x) \end{aligned}$$

同样可有

$$\sim(\exists x)A(x) \Leftrightarrow (\forall x) \sim A(x)$$

上述等价关系推广到无限个体域后仍然是成立的。

量词辖域的扩张及收缩律 设 P 中不出现约束变元 x , 则有

$$\begin{aligned} (\forall x)A(x) \vee P &\Leftrightarrow (A(a_1) \wedge A(a_2) \wedge \cdots \wedge A(a_n)) \vee P \\ &\Leftrightarrow (A(a_1) \vee P) \wedge (A(a_2) \vee P) \wedge \cdots \wedge (A(a_n) \vee P) \Leftrightarrow (\forall x)(A(x) \vee P) \end{aligned}$$

用同样的方法可以证明以下 3 个等价式也成立:

$$(\forall x)A(x) \wedge P \Leftrightarrow (\forall x)(A(x) \wedge P)$$

$$(\exists x)A(x) \vee P \Leftrightarrow (\exists x)(A(x) \vee P)$$

$$(\exists x)A(x) \wedge P \Leftrightarrow (\exists x)(A(x) \wedge P)$$

量词分配律 对任意谓词公式 $A(x)$ 和 $B(x)$ 有

$$(\forall x)(A(x) \wedge B(x)) \Leftrightarrow (A(a_1) \wedge B(a_1)) \wedge (A(a_2) \wedge B(a_2)) \wedge \dots$$

$$\Leftrightarrow (A(a_1) \wedge A(a_2) \wedge \dots) \wedge (B(a_1) \wedge B(a_2) \wedge \dots) \Leftrightarrow (\forall x)A(x) \wedge (\forall x)B(x)$$

即 $(\forall x)$ 对 \wedge 服从分配律。同样有

$$(\exists x)(A(x) \vee B(x)) \Leftrightarrow (\exists x)A(x) \vee (\exists x)B(x)$$

即 $(\exists x)$ 对 \vee 服从分配律。

但是, $(\forall x)$ 对 \vee 及 $(\exists x)$ 对 \wedge 都不服从分配律, 仅满足:

$$(\exists x)(A(x) \wedge B(x)) \Rightarrow (\exists x)A(x) \wedge (\exists x)B(x)$$

$$(\forall x)A(x) \vee (\forall x)B(x) \Rightarrow (\forall x)(A(x) \vee B(x))$$

总结以上讨论, 并用类似的方法, 可得出谓词逻辑中特有的一些重要等价式和蕴含式如下, 其中序号接 3.1.2 小节。

量词转化律:

$$E_{41} \quad \sim(\exists x)A(x) \Rightarrow (\forall x)\sim A(x)$$

$$E_{42} \quad \sim(\forall x)A(x) \Rightarrow (\exists x)\sim A(x)$$

量词辖域扩张及收缩律

$$E_{43} \quad (\forall x)A(x) \vee P \Leftrightarrow (\forall x)(A(x) \vee P)$$

$$E_{44} \quad (\forall x)A(x) \wedge P \Leftrightarrow (\forall x)(A(x) \wedge P)$$

$$E_{45} \quad (\exists x)A(x) \vee P \Leftrightarrow (\exists x)(A(x) \vee P)$$

$$E_{46} \quad (\exists x)A(x) \wedge P \Leftrightarrow (\exists x)(A(x) \wedge P)$$

量词分配律:

$$E_{47} \quad (\exists x)(A(x) \vee B(x)) \Rightarrow (\exists x)A(x) \vee (\exists x)B(x)$$

$$E_{48} \quad (\forall x)(A(x) \wedge B(x)) \Rightarrow (\forall x)A(x) \wedge (\forall x)B(x)$$

其他等价式:

$$E_{49} \quad (\forall x)A(x) \rightarrow B \Leftrightarrow (\exists x)(A(x) \rightarrow B)$$

$$E_{50} \quad (\exists x)A(x) \rightarrow B \Leftrightarrow (\forall x)(A(x) \rightarrow B)$$

$$E_{51} \quad A \rightarrow (\forall x)B(x) \Leftrightarrow (\forall x)(A \rightarrow B(x))$$

$$E_{52} \quad A \rightarrow (\exists x)B(x) \Leftrightarrow (\exists x)(A \rightarrow B(x))$$

$$E_{53} \quad (\exists x)(A(x) \rightarrow B(x)) \Leftrightarrow (\forall x)A(x) \rightarrow (\exists x)B(x)$$

蕴含式:

$$I_{17} \quad (\forall x)A(x) \vee (\forall x)B(x) \Rightarrow (\forall x)(A(x) \vee B(x))$$

$$I_{18} \quad (\exists x)(A(x) \wedge B(x)) \Rightarrow (\exists x)A(x) \wedge (\exists x)B(x)$$

$$I_{19} \quad (\exists x)A(x) \rightarrow (\forall x)B(x) \Rightarrow (\forall x)(A(x) \rightarrow B(x))$$

$$I_{20} \quad (\forall x)(A(x) \rightarrow B(x)) \Rightarrow (\forall x)A(x) \rightarrow (\forall x)B(x)$$

量词次序的交换 从量词的意义出发, 还可给出一组量词交换式:

$$(\forall x)(\forall y)P(x, y) \Leftrightarrow (\forall y)(\forall x)P(x, y)$$

$$(\forall x)(\forall y)P(x, y) \Rightarrow (\exists y)(\forall x)P(x, y)$$

$$(\forall y)(\forall x)P(x, y) \Rightarrow (\exists x)(\forall y)P(x, y)$$

$$(\exists y)(\forall x)P(x, y) \Rightarrow (\forall x)(\exists y)P(x, y)$$

$$(\exists x)(\forall y)P(x, y) \Rightarrow (\forall y)(\exists x)P(x, y)$$

$$(\forall x)(\exists y)P(x, y) \Rightarrow (\exists y)(\exists x)P(x, y)$$

$$(\forall y)(\exists x)P(x, y) \Rightarrow (\exists x)(\exists y)P(x, y)$$

$$(\exists x)(\exists y)P(x, y) \Leftrightarrow (\exists y)(\exists x)P(x, y)$$

3.2.7 谓词逻辑中的推论规则

谓词逻辑是一种比命题逻辑范围更加广泛的形式语言系统。命题逻辑中的推论规则都可以无条件地推广到谓词逻辑中来。此外, 谓词逻辑中还有一些自己独立的推论规则。

约束变元的改名规则 谓词公式中约束变元的名称是无关紧要的, 通常认为 $(\forall x)P(x)$ 和 $(\forall y)P(y)$ 具有相同的意义, 因此需要时可以改变约束变元的名称。但必须遵守以下改名规则:

(1) 要改名的变元应是某量词作用范围内的变元, 且应同时更改该变元在此量词辖域内的所有约束出现, 而公式的其余部分不变;

(2) 新的变元符号应是此量词辖域内原先没有的。

自由变元的代入规则 自由变元也可以改名, 但必须遵守以下规则:

(1) 要改自由变元 x 的名, 必改 x 在公式中的每个自由出现;

(2) 新变元不应在原公式中以任何约束形式出现。

命题变元的代换规则 用任一谓词公式 A_i 替换永真公式 B 中某一命题变元 P_i 的所有出现, 形成的新公式 B' 仍然是永真式 (但在 A_i 的个体变元中, 不应有 B 中的约束变元出现), 并有 $B \Rightarrow B'$ 。

取代规则 设 $A'(x_1, x_2, \dots, x_n) \Leftrightarrow B(x_1, x_2, \dots, x_n)$ 都是含 n 个自由变元的谓词公式, 且 A' 是 A 的子式。若在 A 中用 B' 取代 A' 的一处或多处出现后所得的新公式是 B , 则有 $A \Leftrightarrow B$ 。如果 A 为永真式, 则 B 也是永真式。

关于量词的增删, 还有 4 条规则:

(1) 全称规定规则 US

从 $(\forall x)A(x)$ 可得出结论 $A(y)$, 其中 y 是个体域中任一个体, 即

$$(\forall x)A(x) \Rightarrow A(y)$$

使用 US 规则的条件是, 对于 y , 公式 $A(x)$ 必须是自由的。根据 US 规则, 在推论过程中可以移去全称量词。

(2) 存在规定规则 ES

从 $(\exists x)A(x)$ 可得到结论 $A(y)$, 其中 y 是个体域中某一个特殊个体, 即

$$(\exists x)A(x) \Rightarrow A(y)$$

使用 ES 规则的条件是, y 必须是在前面没有出现过的, 以免发生混淆。这就是说, 在给定的所有前提中, y 都不是自由的; 在居先的任何推导步骤上, y 也不是自由的。根据 ES 规则, 在推论中可以移去存在量词。

(3) 存在推广规则 EG

从 $A(x)$ 可得出结论 $(\exists y) A(y)$, 其中 x 是个体域中某一个个体, 即

$$A(x) \Rightarrow (\exists y)A(y)$$

使用 EG 规则的条件是, 对于 y , 公式 $A(x)$ 必须是自由的。根据 EG 规则, 在推论过程中可以附上存在量词。

(4) 全称推广规则 UG

从 $A(x)$ 可得出结论 $(\forall y) A(y)$, 其中 x 应是个体域中任一个体, 即

$$A(x) \Rightarrow (\forall y)A(y)$$

使用 UG 规则的条件: 在任何给定的前提中, x 都不是自由的; 在使用 ES 规则而得到的一个居先步骤上, 如果 x 是自由的, 则由于使用 ES 规则而引入的任何新变元在 $A(x)$ 中都不是自由出现。根据 UG 规则, 在推论过程中可以附上全称量词。

3.2.8 谓词公式的范式与斯柯林标准形

命题逻辑中的 4 种范式都可以直接推广到谓词逻辑中来。只要把原子命题公式换成原子谓词公式即可。此外, 根据量词在公式中出现的情况不同, 又可分为前束范式和斯柯林 (Skolem) 范式。

前束范式 设有一谓词公式 F , 如果其中所有量词均非否定地出现在公式的最前面, 且它们的辖域为整个公式, 则称 F 为前束范式。例如,

$$(\forall x)(\forall y)(\exists z)(P(x, y) \vee Q(x, z) \wedge R(x, y, z))$$

是前束范式。

任一公式都可以化为与之等价的前束范式。其方法如下:

(1) 消去公式中的联结词 和 \rightarrow (E_{38}, E_{39});

(2) 将公式内的否定符号深入到谓词变元 ($E_{11}, E_{12}, E_{47}, E_{48}$), 并化简到谓词变元前最多只有一个 \sim (E_{10});

(3) 利用改名、代入规则使所有约束变元均不同, 且使自由变元与约束变元也不同;

(4) 扩充量词的辖域至整个公式 ($E_{43}, E_{44}, E_{45}, E_{46}$)。

例 6: 将公式 $((\forall x)P(x) \vee (\exists y)R(y)) \rightarrow (\forall x)F(x)$ 化为前束范式。

解:

$$\begin{aligned} & ((\forall x)P(x) \vee (\exists y)R(y)) \rightarrow (\forall x)F(x) \\ \Leftrightarrow & \sim((\forall x)P(x) \vee (\exists y)R(y)) \vee (\forall x)F(x) \\ \Leftrightarrow & (\sim(\forall x)P(x) \wedge \sim(\exists y)R(y)) \vee (\forall x)F(x) \\ \Leftrightarrow & ((\exists x) \sim P(x) \wedge (\forall y) \sim R(y)) \vee (\forall x)F(x) \\ \Leftrightarrow & ((\exists x) \sim P(x) \wedge (\forall y) \sim R(y)) \vee (\forall z)F(z) \\ \Leftrightarrow & (\exists x)(\forall y)(\forall z)((\sim P(x) \wedge \sim R(y)) \vee F(x)) \end{aligned}$$

斯柯林范式 如果前束范式中所有的存在量词均在全称量词之前，则称这种形式为斯柯林范式。例如，

$$(\exists x)(\exists z)(\forall y)(P(x,y) \vee Q(y,z) \wedge R(y))$$

是斯柯林范式。

任何一个公式都可以化为与之等价的斯柯林范式，其方法如下：

- (1) 先将给定公式化为前束范式；
- (2) 将前束范式中的所有自由变元用全称量词约束 (UG)；
- (3) 若经上述改造后的公式 A 中，第 1 个量词不是存在量词，则可以将 A 等价变换成如下形式：

$$(\exists u)(A \wedge (G(u) \vee \neg G(u)))$$

其中， u 是 A 中没有的个体变元；

- (4) 如果前束是由 n 个存在量词开始，然后是 m 个全称量词，后面还跟有存在量词，则可以利用下述等价式将这些全称量词逐一移到存在量词之后：

$$\begin{aligned} & (\exists x_1) \cdots (\exists x_n)(\forall y)P(x_1, \cdots, x_n, y) \\ & \Leftrightarrow (\exists x_1) \cdots (\exists x_n)(\exists y)((P(x_1, \cdots, x_n, y) \wedge \neg H(x_1, \cdots, x_n, y)) \vee (\forall z)H(x_1, \cdots, x_n, z)) \end{aligned}$$

其中， $P(x_1, x_2, \cdots, x_n, y)$ 是一个前束范式，它仅含有 x_1, x_2, \cdots, x_n 和 y 等 $n+1$ 个自由变元。 H 是不出现于 P 内的 $n+1$ 元谓词。把等价式右边整理成前束范式，它的前束将是一个以 $(\exists x_1) \cdots (\exists x_n)(\exists y)$ 开头，后面跟有 P 中的全称量词和存在量词，最后是 $(\forall z)$ 。如果作用 m 次，就可将存在量词前面的 m 个全称量词全部移到存在量词之后去。

斯柯林范式比前束范式更优越，它将任一公式分为 3 部分：存在量词序列、全称量词序列和不含量词的谓词公式。这大大方便了对谓词公式的研究。

斯柯林标准形

设有一个公式 F 的前束范式为

$$(Q_1x_1) \cdots (Q_nx_n) M$$

其中， M 是合取范式， $((Q_1x_1) \cdots (Q_nx_n))$ 是量词序列。设 $Q_r (1 \leq r \leq n)$ 是量词序列中的一个存在量词。

- (1) 如果没有全称量词出现在 Q_r 之前，就选择一个 M 中未出现过的常量 a ，代替所有在 M 中出现的 x_r ；

- (2) 如果 Q_{s_1}, \cdots, Q_{s_m} 是出现在 Q_r 之前的所有全称量词，其中 $1 \leq s_1 < s_2 < \cdots < s_m \leq r$ ，则选择一个 M 中没有出现过的 m 元函数符号 f ，用 $f(x_{s_1}, x_{s_2}, \cdots, x_{s_m})$ 代替 M 中出现的所有 x_r ，并在前束中消去 (Q_rx_r) 。

用上述方法除去公式 F 的前束中的所有存在量词后得到的最后公式，叫做公式 F 的斯柯林标准形，简称标准形。用来代替 x_r 的 a 和 f 称为斯柯林函数。标准形中量词后的内容为母式。

例如， $F = (\forall x)(\exists y)(\exists z)((\sim P(x,y) \vee R(x,y,z)) \wedge (Q(x,z) \vee R(x,y,z)))$ 的斯柯林标准形为

$$(\forall x)((\sim P(x, f(x)) \vee R(x, f(x), g(x))) \wedge (Q(x, g(x)) \vee R(x, f(x), g(x))))$$

子句是一些文字的析取。由于谓词公式的标准形的母式已经是合取范式，从而母式的每个合取项都是一个子句，于是可以说母式是由一些子句的合取构成的。进而略去标准形

中的全称量词，以逗号“,”代替合取符号 \wedge ，便得到公式的子句集，故对于谓词公式的讨论将对子句集的讨论替代，因为子句集是一种更为简单的标准形式。它也是学习谓词逻辑归结原理的基础。

3.3 产生式系统

产生式系统（production system）是 1943 年波斯特（Post）提出的一种计算形式体系里所使用的术语，类似文法的规则，对符号串做替换运算。到了 20 世纪 60 年代，产生式系统成为认知心理学研究人类心理活动中信息加工过程的基础，并用它来建立人类认识的模型。现在，产生式系统在人工智能领域内，无论在理论上和应用上都经历了很大的发展，所以现在的产生式系统和波斯特的系统已很不相同。它已发展成为人工智能系统中最典型最普遍的一种结构，例如，目前大多数专家系统都采用产生式系统的结构来建造。采用产生式系统作为 AI 系统的主要结构，原因有两点：第一，用产生式系统结构求解问题的过程和人类求解问题时的思维过程很相像，因而可以用它来模拟人类求解问题时的思维过程；第二，可以把产生式系统作为 AI 系统的基本结构单元或基本模式看待，就好像是积木块一样，因而研究产生式系统的基本问题就具有一般意义。

3.3.1 产生式系统的基本组成

一个高效的人工智能系统需要大量的有关知识作为背景，知识可以分为 3 部分：叙述性知识、过程性知识和控制性知识。AI 系统的任务就是要把这 3 部分知识有效地组织起来，形成一个系统，以便实现问题求解过程的自动化。

产生式系统中，与这 3 方面的知识相对应，也包含了 3 个基本组成部分：综合数据库（global database）、一组产生式规则（set of rules）和一个控制策略（control strategies）。它们之间的关系如图 3.1 所示。

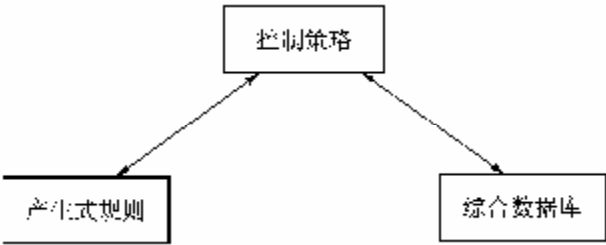


图 3.1 产生式系统的基本组成

综合数据库是产生式系统所使用的主要数据结构，它用来表述问题状态或有关事实，即它含有所求解问题的信息，其中有些部分可以是不变的，有些部分则可能只与当前问题的解有关。人们可以根据问题的性质，用适当的方法来构造综合数据库的信息。综合数据库对应着叙述性知识，相当于人脑的短期记忆功能。

产生式规则集是作用在全局数据库上的一些规则（算子、操作）的集合，每条规则都有一定的条件，若全局数据库中的内容满足这个条件，就可以调用这条规则，执行规则的结果会改变全局数据库中的内容。产生式规则对应着过程性知识，相当于人脑的长期记忆功能。产生式规则的一般形式如下：

条件→行动

或

前提→结论

即表示为

if ... then ...

的形式。其中左半部确定了该规则可应用的先决条件，右半部描述了应用这条规则所采取的行动或得出的结论。一条产生式规则满足了先决条件之后就可对综合数据库进行操作，使其发生变化。如综合数据库代表当前状态，则应用规则后就使状态发生变化，生成新状态。

控制系统或控制策略是负责选择规则的决策系统，即决定了问题求解过程的推理路线。当数据库满足结束条件时，系统就应停止运行。还要使系统在求解过程中记住应用过的规则序列，以便最终能给出解的路径。它对应着控制性知识。

通常从选择规则到执行操作分 3 步：匹配、冲突消解和操作。

（1）匹配

在这一步，把当前数据库和规则的条件部分相匹配。如果两者完全匹配，则把这条规则称为触发的规则。当按规则的操作去执行时，把这条规则称为被启用的规则。被触发的规则不一定总是被启用的规则，因为可能同时有几条规则的条件部分被满足，这就要在解决冲突步骤中来解决这个问题。在复杂的情况下，在数据库和规则的条件部分之间可能要近似匹配。

（2）冲突消解

当有一个以上的规则条件部分和当前数据库相匹配时，就需要决定首先使用哪条规则，这称为冲突消解。例如，设有以下两条规则：

规则 R1	IF	fourth dawn short yardage
	THEN	punt
规则 R2	IF	fourth dawn short yardage within 30 yards(from the goal line)
	THEN	field goal

这是两条关于美式足球的规则。规则 R1 规定，进攻这一方如果在前 3 次进攻中前进的距离少于 10 码（short yardage），那么在第 4 次进攻（fourth dawn）时，可以踢凌空球（punt）。规则 R2 规定，如果进攻这一方在前 3 次进攻中前进的距离少于 10 码，而进攻的位置又在离对方球门线 30 码距离之内（within 30 yards from the goal line），那就可以射门（field goal）。

如果当前数据库包含事实 short yardage 以及 within 30 yards，则上述两条规则都被触发，这就需要用冲突消解策略来决定首先使用哪条规则，也就是确定规则启用顺序。一般

的冲突消解策略有以下几种：

① 专一性排序

如果某一规则的条件部分规定的情况，比另一规则的条件部分所规定的情况更为专门，则这条规则有较高的优先级。按此方法，上述例子中的规则 R2 有较高的优先级。

② 规则排序

如果规则编排的顺序就表示了启用的优先级，则称为规则排序。按此方法，在上述例子中 R1 的优先级将比 R2 高。因为按规则排序的次序 R1 排在前面。

③ 数据排序

如果把规则条件部分的所有条件按优先级次序编排起来。运行时首先使用在条件部分包含较高优先级数据库的规则。例如，在上述例子中如果“进攻的位置在离对方球门线 30 码距离之内”，这个条件具有较高的优先级，那就首先应选用 R2 规则。

④ 规模排序

这种方法按规则的条件部分的规模排列优先级。优先使用被满足的条件较多的规则。

⑤ 就近排序

这种方法把最近使用的规则放在最优先的位置。这和人类的行为有相似之处。如果某一规则经常被使用，则人们倾向于更多地使用这条规则。

⑥ 上下文限制

这种方法把产生式规则按它们所描述的上下文分组，也就是说按上下文对规则分组。在某种上下文条件下，只能从与其相对应的那组规则中选择应用的规则。

不同的系统，使用上述这些策略的不同组合，如何选择冲突解决策略完全是启发式的。

(3) 操作

操作就是执行规则的操作部分。经过操作以后，当前数据库将被修改。然后，其他的规则有可能被使用。

3.3.2 产生式系统的基本过程

用产生式系统求解问题的过程可用下列算法来描述：

(1) DATA ← 初始数据库

(2) until DATA 满足终止条件, do

(3) begin

(4) 在规则集中选择能作用到 DATA 上的规则 R

(5) DATA ← R 作用到 DATA 上的结果

(6) end

这个过程是不确定的，因为在 (4) 中没有明确规定如何挑选一条可用的规则。关于选择规则的方法，属于控制系统中搜索策略研究的范围。有效的搜索策略，要求对被解问题有足够的先验知识，以便在 (4) 中选出最合适的规则。但在多数人工智能系统中，搜索策略中包含的先验知识不足以保证在每次通过 (4) 时都选出最佳规则，所以，在产生式系统中实现的仍然是一个搜索过程，它要对一系列的规则进行试探，直到发现某一规则序列，

使全局数据库满足终止条件为止。

通过上述过程可以看出,产生式系统与一般分级组织的计算机软件系统比较,具有以下特点:

- (1) 全局数据库的内容可以为所有的规则所访问,没有任何部分是专为某一规则建立的,这种特性便于模仿智能行为中的强数据驱动性。
- (2) 规则本身不能调用其他规则。规则之间的联系必须通过全局数据库进行。
- (3) 全局数据库、规则和控制系統之间相对独立,这种积木式结构便于整个系统增加和修改知识。这些特点对于建立一个大型人工智能系统是十分有用的。

3.3.3 基于产生式系统的具体问题建模

这里举例说明如何用产生式系统来描述或表示求解的问题,即如何对具体的问题建立起产生式系统的描述,以及用产生式系统求解问题的基本思想。

八数码游戏问题 (Eight-Puzzle): 在 3×3 组成的九宫格棋盘上,每个将牌都刻有 1~8 中的某个数码。棋盘中留有一个空格,允许其周围的某一个将牌向空格移动,这样通过移动将牌就可以不断改变将牌的布局。这种游戏求解的问题是:给定一种初始的将牌布局或结构(称初始状态)和一个目标布局(称目标状态),问如何移动将牌,实现从初始状态向目标状态的转变。问题的解答其实就是给出一个合法的走步序列。

要用产生式系统来求解这个问题,首先必须建立起问题的产生式系统描述,即把问题的叙述转化为产生式系统的 3 个组成部分,在 AI 中通常称为问题的表示。一般来说,问题可有多种表示方式,而选择一种较好的表示是运用人工智能技术解决实际问题首先要考虑的,而且要有一定的技巧。

设给定的具体问题如图 3.2 所示。

(1) 综合数据库:这里是要选择一种数据结构来表示将牌的布局。通常可用来表示综合数据库的数据结构有字符串、向量、集合、数组、树、表格、文件等。对八数码问题,选用二维数组来表示将牌的布局很直观,因此该问题的综合数据库可用如下形式表示:

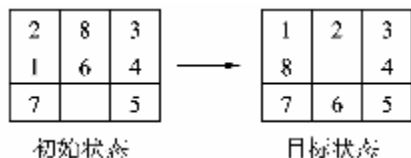


图 3.2 八数码游戏实例

(S_{ij}) 其中 $1 \leq i, j \leq 3, S_{ij} \in \{0, 1, \dots, 8\}$, 且 S_{ij} 互不相等

这样每个具体的矩阵就可表示一个棋局状态。对八数码游戏,显然共有 $9! = 362\,880$ 个状态。所有可能的状态集合就构成该问题的状态空间或问题空间。可以证明八数码问题的实际问题空间只有 $1/2 \times 9! = 181\,440$ 。

(2) 规则集合:移动一块将牌(即走一步)就使状态发生转变。改变状态有 4 种走法:空格左移、空格上移、空格右移、空格下移。这 4 种走法可用 4 条产生式规则来模拟,应用每条规则都应满足一定的条件。于是规则集可形式化表示如下:

设 S_{ij} 记矩阵第 i 行第 j 列的数码, i_0, j_0 记空格所在的行、列数值,即 $S_{i_0 j_0} = 0$, 则
 if $j_0 - 1 \geq 1$ then $S_{i_0 j_0} := S_{i_0 (j_0 - 1)}, S_{i_0 (j_0 - 1)} := 0; \quad (S_{i_0 j_0} \text{ 向左})$

if $i_0-1 \geq 1$ then $Si_0j_0 := S(i_0-1)j_0$, $S(i_0-1)j_0 := 0$; (Si_0j_0 向上)
 if $j_0+1 \leq 3$ then $Si_0j_0 := Si_0(j_0+1)$, $Si_0(j_0+1) := 0$; (Si_0j_0 向右)
 if $i_0+1 \leq 3$ then $Si_0j_0 := S(i_0+1)j_0$, $S(i_0+1)j_0 := 0$; (Si_0j_0 向下)

(3) 搜索策略：是从规则集中选取规则并作用于状态的一种广义选取函数。确定某种策略后，以算法的形式给出。在建立产生式系统描述时，还要给出初始状态和目标条件，具体说明所求解的问题。产生式系统中控制策略的作用就是从初始状态出发，寻找一个满足一定条件的问题状态。对该问题，初始状态可表示为

目标描述可表示为

它是一种显式表示的描述。更一般的情况可以是规定达到目标的某个真或假

条件的描述，如规定要达到第①行将牌数码总和为6的状态。显然这样一个条件，隐含地确定了目标是一个状态的子集——目标集。目标条件也是产生式系统结束条件的基础。

建立了产生式系统描述后，通过控制策略，可求得实现目标的一个走步序列（即规则序列），这就是所谓的问题的解，如走步序列（上、上、左、下、右）就是一个解。这个解序列是根据控制系统记住搜索目标过程中用过的所有规则而构造出来的。

3.3.4 产生式系统的类型

根据控制系统、规则、综合数据库的内容和应用的不同，产生式系统可以分为几种不同的类型，尼尔逊根据控制策略的不同提出了如下的产生式系统分类体系：

(1) 按搜索策略划分

按搜索策略，产生式系统可分为不可撤回的产生式系统和试探性的产生式系统。

不可撤回的产生式系统是指规则使用后，不允许回过头来重新选用其他规则。如爬山法，只考虑选有最大增量的规则向上爬，不允许退下来，故有可能遇到局部极值。换句话说，这种方式是利用问题给出的局部知识来决定如何选取规则的。根据当前可靠的局部知识选一条可应用的规则，并作用于当前综合数据库，接着再根据新的状态继续选取规则，搜索过程一直进行下去。不必考虑撤回用过的规则，这是由于在搜索过程中如果能有效利用局部知识，即使使用了一条不理想的规则，也不妨碍下一步选得另一条更合适的规则。显然，这种策略具有控制简单的优点。

试探性的产生式系统指规则使用后，允许返回出发点重新选择其他规则。试探性的产生式系统又分为两类：回溯式产生式系统和图搜索式产生式系统。

回溯式产生式系统：当搜索遇到困难时，可返回再选其他规则。例如，有界深度优先搜索。

图搜索式产生式系统：它同时掌握若干规则序列的效果，从中寻找问题的答案。为避免循环，通常采用树搜索法，例如，广度优先搜索。

(2) 按搜索方向划分

一般是从初始状态向着目标方向进行搜索,如果规则可以逆向运用,也可以从目标状态向着初始状态方向进行搜索,或者双向同时进行搜索。这样就形成了3种不同方向的产生式系统:正向产生式系统、逆向产生式系统、双向产生式系统。

正向产生式系统是从初始状态出发,朝着目标状态这个方向来使用规则,即正推的方式工作,称这些规则为F规则。反过来如果选取目标描述作为初始综合数据库逆向进行求解,即从目标状态出发,反方向一步一步朝着初始状态方向求解,则称为逆向产生式系统,逆向应用的规则称为B规则。若以双向搜索方式去求解问题,则称为双向产生式系统。这时必须把状态描述和目标描述合并构成综合数据库,F规则只适用于状态描述部分,B规则只适用于目标描述部分。这种类型的搜索,其控制策略所使用的结束条件要表示成综合数据库中状态描述部分与目标描述部分之间某种形式的匹配条件,而且搜索时还要决定每段上要选用F规则还是B规则。

(3) 其他分类

除一般类型的产生式系统外,还有两种特殊类型的产生式系统,即可交换的产生式系统与可分解的产生式系统。

可交换的产生式系统:各规则的选用次序不重要。

可分解的产生式系统:初始数据库可分解为若干独立部分,并能用规则单独对各部分进行处理,终止条件也可以相应被分解。

3.3.5 产生式系统的搜索策略

搜索策略的主要任务是确定如何选取规则的方式。有两种基本的方式:一种是不考虑给定问题所具有的特定知识,系统根据事先确定好的某种固定排序,依次调用规则或随机调用规则,这实际上就是盲目搜索方法,一般统称为无信息引导的搜索策略。另一种是考虑问题领域可应用的知识,动态地确定规则的排序,优先调用较合适的规则使用,这就是通常的启发式搜索策略或有信息引导的搜索策略。到目前为止,AI领域中已提出许多具体的搜索方法。这里主要讨论用不可撤回策略和回溯控制策略求解八数码问题的方法。

1. 不可撤回的控制方法

对于不可撤回的控制方法,首先要建立一个描述综合数据库变化的函数,如果这个函数具有单极值,且这个极值对应的状态就是目标,则不可撤回的控制策略就是选择使函数值发生最大增长变化的那条规则来作用于综合数据库,如此循环下去,直到没有规则使函数值继续增长,这时函数值取得最大值,满足结束条件。

下面通过八数码游戏的例子加以说明。用“不在位”将牌个数并取其负值作为状态描述的函数 $-W(n)$ (“不在位”将牌个数是指当前状态与目标状态对应位置逐一比较后有差异的将牌总个数,用 $W(n)$ 表示,其中 n 表示任一状态),例如,图3.2中的初始状态,其函数值是-4,而目标状态的函数值是0。用这样定义的函数就能计算出任一状态的函数值来。

从初始状态出发,看如何应用这个函数来选取规则。对初始状态,有3条可应用规则,空格向左和空格向右这两条规则生成新的状态,其 $-W(n)$ 均为-5,空格向上所得新状态,其

$-W(n)$ 为-3，比较后看出这条规则可获得函数值的最大增长，所以产生式系统就选取这条规则来应用。按此一步一步进行下去，直至产生式系统结束时就可获得解。图 3.3 表示出求解过程所出现的状态序列，图中矩阵下面的数字就是该状态的函数值（或称爬山函数值）。从图中还可以看出，沿着状态变化路径，出现函数值不增加的情况，就是说出现了没有一条合适的规则能使函数值增加，这时就要任选一条函数值不减小的规则来应用，如果不存在这样的规则，则过程停止。

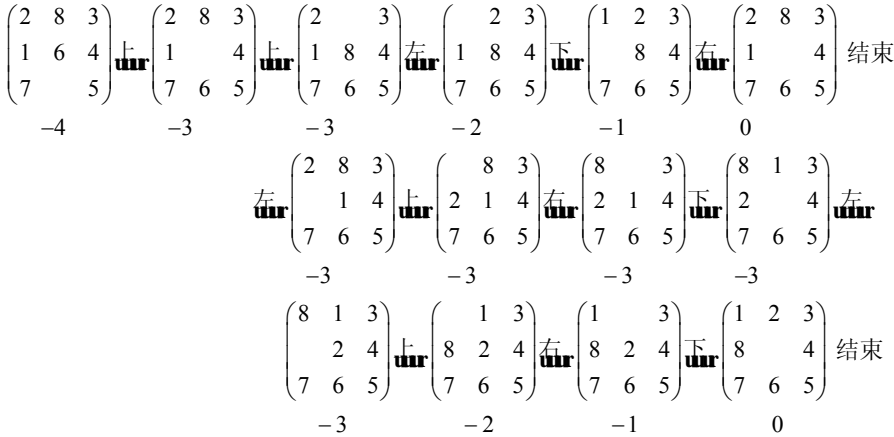


图 3.3 八数码游戏各状态的爬山函数值

从图 3.3 中所示的情况来看，用不可撤回策略能找到一条通往目标的路径。然而一般来说，爬山函数会有多个局部极大值的情况，这样就会破坏爬山法找到真正的目标。例如，初始状态和目标状态分别为

$$\begin{pmatrix} 1 & 2 & 5 \\ & 7 & 4 \\ 8 & 6 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ & 7 & 4 \\ 8 & 6 & 5 \end{pmatrix}$$

时，任意一条可应用于初始状态的规则，都会使 $-W(n)$ 下降，这相当于初始状态的描述函数值处于局部极大值上，搜索过程停止不前，找不到代表目标的全局极大值。

根据以上讨论看出，对 AI 感兴趣的一些问题，使用不可撤回的策略，虽然其控制简单，然而它不可能对任何状态总能选得最优的规则，甚至有时很难对给定问题构造出任何情况下都能通用的简单爬山函数（该爬山函数应该不具有多极值或“平顶”等情况）。因而不可撤回的方式具有一定的局限性。

2. 回溯控制方法

在问题求解过程中，有时会发现用一条不适合的规则，会阻挠或拖延达到目标的过程。在这种情况下，需要有这样的控制策略：先试一试某一条规则，如果以后发现这条规则不合适，允许退回去，另选一条规则来试。

使用回溯策略首要的问题是要研究在什么情况下应该回溯，即回溯条件；其次就是如何利用有用的知识进行规则排序，以减少回溯次数。下面仍用八数码问题来讨论回溯条件，即搜索过程，有关利用知识选取规则的问题在后面介绍。这里选择应用的规则采用事先确

定的固定排序依次选取。例如，以左、上、右、下这种顺序来选取规则。

对八数码问题，回溯应发生在以下 3 种情况：

- (1) 新生成的状态在通向目标状态的路径上已经出现过；
- (2) 从初始状态开始，应用的规则数目达到所规定的数目之后还未找到目标状态（这一组规则的数目实际上就是搜索深度范围所规定的）；
- (3) 对当前状态来说，再没有可应用的规则。

图 3.4 表示出回溯策略应用于八数码游戏时的一部分搜索图，这里规定搜索的深度到第 6 层，即用了 6 条规则后仍没有找到目标就要回溯到上一层。显然 6 层以内的所有路径都会被搜索到，因此对这个问题一定能找到解。然而对一般情况，深度设置太浅时，有可能找不到解，设置太深有可能导致回溯次数聚增，因而应根据实际情况来规定搜索范围，先设置适中的深度搜索，失败后再逐步加深。

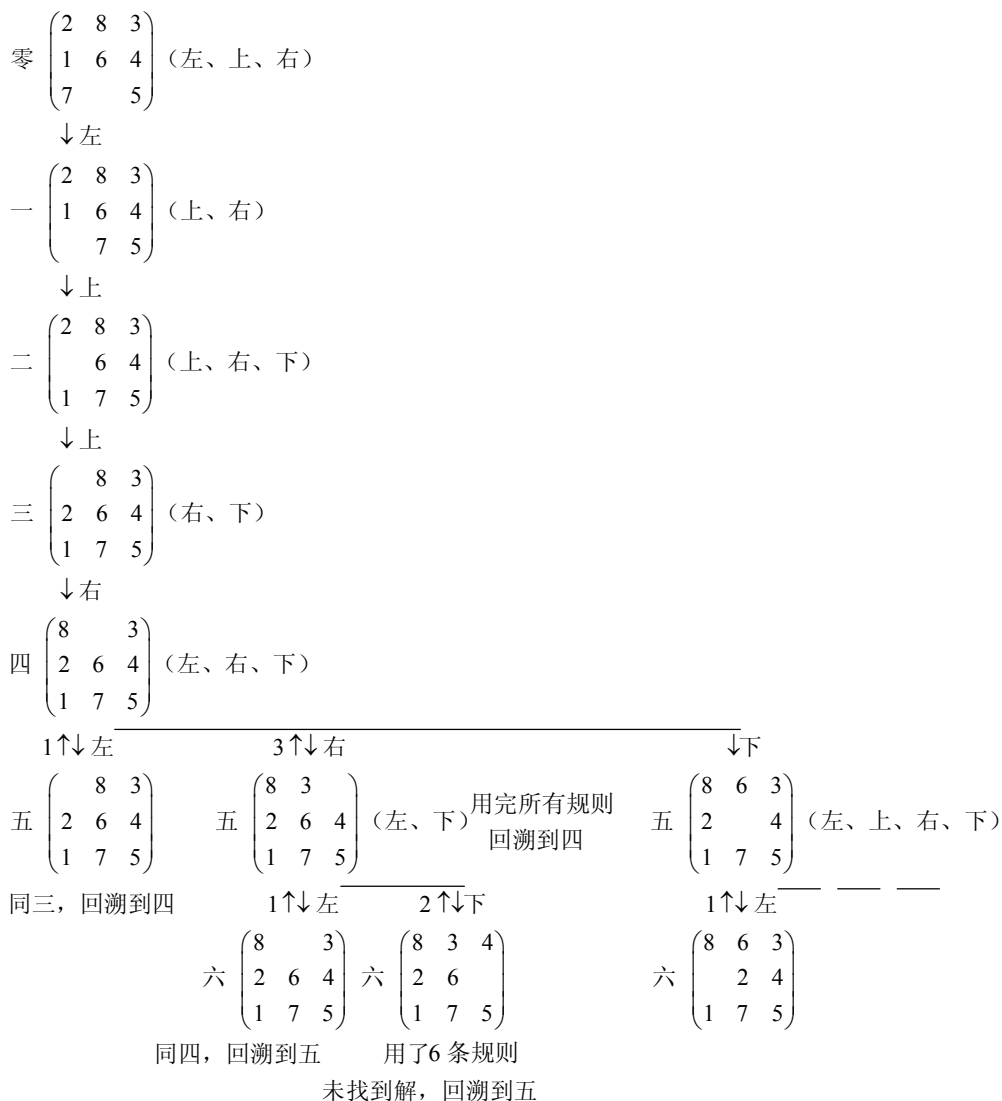


图 3.4 八数码游戏回溯控制方式

回溯过程是一种可试探的方法。从形式上看,不论是否存在对选择规则有用的知识,都可以采用这种策略。即如果没有有用的知识来引导规则选取,规则可按任意方式(固定排序或随机)选取;如果有好的选择规则的知识可用,那么用这种知识来引导规则的选取,就会减少盲目性,降低回溯次数,甚至不回溯就能找到解,一般来说这有利于提高效率。此外由于引入回溯机理,可以避免陷入局部极大值的情况,继续寻找其他达到目标的路径。

3.3.6 两种典型的产生式系统

1. 回溯式产生式系统

对于搜索量小的问题,回溯式策略往往是完备和有效的,它有易于实现、占用存储量小等优点。下面来说明这种产生式系统的工作原理。整个系统可用一个递归过程 BACKTRACK 来实现。它是一种局部试探性搜索系统。

给定一个待求解的问题后,首先利用知识表示技术,建立该问题的状态描述(全局数据库 DATA)和操作描述(该问题的规则集)。其他各种知识反映在下述谓词和函数中。

TERM(DATA): 判 DATA 是否满足终止条件的谓词;

DEADEND(DATA): 判 DATA 是否满足失败条件的谓词;

APPRULES(DATA): 选取可作用在 DATA 上的所有操作并排序的函数;

R(DATA): 求操作 R 作用在 DATA 上面所产生的新数据库的函数。

此外还有几个与符号处理有关的谓词和函数。

NULL(RULES): 判可用规则表空否的谓词,空表用 NIL 表示;

FIRST(RULES): 取可用规则表中第 1 个元素的函数;

TAIL(RULES): 在可用规则表中除去第 1 个元素的函数;

CONS(R, PATH): 把元素 R 加入到路径表(PATH)的最前面的函数。

有了这些函数和谓词后,就可以建立下述递归过程 BACKTRACK。

递归过程 BACKTRACK(DATA):

(1) if TERM(DATA),return NIL (按成功返回);

(2) if DEADEND(DATA),return FAIL (按失败返回);

(3) RULES APPRULES(DATA);

(4) LOOP: if NULL(RULES),return FAIL;

(5) $R \leftarrow \text{FIRST}(\text{RULES})$;

(6) RULES TAIL(RULES);

(7) RDATA R(DATA);

(8) PATH BACKTRACK(RDATA);

(9) if PATH=FAIL, go LOOP(重选其他规则);

(10) return CONS(R, PATH)。

上述递归过程在(8)处逐层深入下去,直到 DATA 变成(1)或(2)可以判断的本原问题为止。(10)和(8)可以引导过程逐层回升,通过(10)建立起来的规则序列就是

问题的解。

本过程有可能永不终止，为克服这一缺点，可以采用递归深度限制；记忆一部分已产生过的数据库，防止死循环。采用以上步骤，本过程可以成为一种算法。

为提高效率，在（3）中可以运用各种启发信息来安排规则的先后顺序。

例 7：四皇后问题。在 4×4 的棋盘上放 4 个皇后，要求任两个皇后都不能在同一行，或同一列，或同一对角线上，如图 3.5（c）所示。

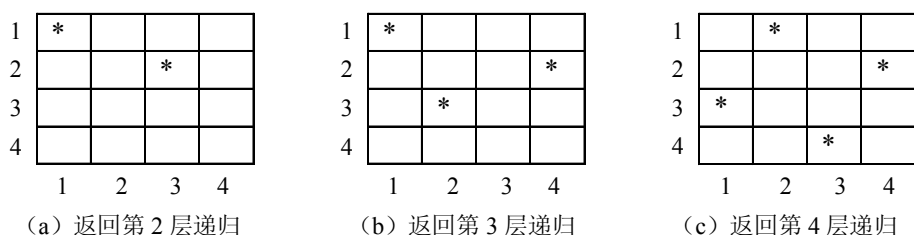


图 3.5 四皇后问题求解过程

解：用递归过程 BACKTRACK 可以求解这个问题。

全局数据库：用 A_{ij} （其中， $1 \leq i \leq 4$ ， $1 \leq j \leq 4$ ）表示出现在位置 (i,j) 的皇后。用已出现的皇后序列表示全局数据库。例如，初始数据库是“（ ）”，中间状态是“（已出现的皇后序列）”，如 (A_{uv}, A_{xy}) 等。

规则集： $\{R_{ij}\}$ （其中， $1 \leq i \leq 4$ ， $1 \leq j \leq 4$ ）。

使用 R_{ij} 的条件是 $i=1$ ，棋盘上无皇后时可以使用； $1 < i \leq 4$ ，第 $i-1$ 行有一皇后时可以使用。

R_{ij} 作用的结果：数据库中增加皇后 A_{ij} 。

控制系统：按自然顺序选择规则。

本产生式系统工作的过程如下：

第 1 层递归

DATA=NIL

RULES=(R11,R12,R13,R14)

R=R11, RULES'=(R12,R13,R14)

RDATA=(A11)

第 2 层递归

DATA=(A11)

RULES=(R23,R24)

R=R23, RULES'=(R24)

RDATA=(A11,A23)

第 3 层递归

DATA=(A11,A23)

RULES=NIL 失败，返回第 2 层递归（如图 3.5（a）所示），返回值是 FAIL。

第 2 层递归

DATA= (A11)

RULES=(R24)

R=R24, RULES'=NIL

RDATA=(A11,A24)

第3层递归

DATA=(A11,A24)

RULES=(R32)

R=R32, RULES'=NIL

RDATA=(A11,A24,A32)

第4层递归

DATA=(A11,A24,A32)

RULES=NIL 失败, 返回第3层递归 (如图3.5 (b) 所示), 返回值是 FAIL。

第3层递归

DATA=(A11,A24)

RULES=NIL 失败, 返回第2层递归, 返回值是 FAIL。

第2层递归

DATA=(A11)

RULES=NIL 失败, 返回第1层递归, 返回值是 FAIL。

第1层递归

DATA= ()

RULES=(R12,R13,R14)

R=R12, RULES'=(R12,R14)

RDATA=(A12)

第2层递归

DATA=(A12)

RULES=(R24)

R=R24, RULES'=NIL

RDATA=(A12,A24)

第3层递归

DATA=(A12,A24)

RULES=(R31)

R=R31, RULES'=NIL

RDATA=(A12,A24,A31)

第4层递归

DATA=(A12,A24,A31)

RULES=(R43)

R=R43, RULES'=NIL

RDATA=(A12,A24,A31,A43)

第5层递归

DATA=(A12,A24,A31,A43)

TERM(DATA)=T 成功, 返回第 4 层递归 (如图 3.5 (c) 所示), 返回值是 NIL。

从第 5 层向上, 一层层退出递归, 每上升一层, 在路径表中加入一个有效的规则 R, 最后可得问题的解:

PATH=(R12,R24,R31,R43)

2. 可分解的产生式系统

可分解的产生式系统是求解与/或图问题的系统, 它也是一种特殊的产生式系统, 其特点是初始数据库可以分解为一些相互独立的部分, 并能用规则对各部分进行处理, 终止条件也可以相应被分解。可分解的产生式系统可用下述基本过程来描述。

过程 SPLIT:

- (1) DATA \leftarrow 初始数据库;
- (2) {Di} \leftarrow DATA 的分解式(现在单独的 Di 可以被看作是独立的数据库);
- (3) until 所有的 {Di} 满足终止条件为止, do
- (4) begin
- (5) 从不满足终止条件的状态集合 {Di} 中选出 D*;
- (6) 从 {Di} 中抹去 D*;
- (7) 从规则集选出可以作用在 D* 上的一个规则 R;
- (8) D \leftarrow R 作用到 D* 上所得的结果;
- (9) {di} \leftarrow D 的分解式;
- (10) 把 {di} 合并到 {Di} 中去;
- (11) end.

在 SPLIT 系统中, 控制策略在 (5) 和 (7) 发挥作用: 在 (5), 它决定从 {Di} 中选出 D* 的最好次序; 在 (7), 它决定选出作用在 D* 上的最好规则 R。但根据 (3) 的要求, {Di} 中的全部元素必须选到。

有两种方式安排 {Di} 中的顺序, ① 按产生的自然顺序排序; ② 在工作过程中动态地排序。

例 8: 有一个产生式系统, 它的初始数据库为 (C,B,Z), 产生式规则如下:

R1: C \Rightarrow D,L

R2: C \Rightarrow B,M

R3: B \Rightarrow M,M

R4: Z \Rightarrow B,B,M

终止条件为: 数据库中的元素全部为 M。

用图搜索产生式系统可以解决这个问题, 但它会产生许多等价的解, 造成很大浪费, 如图 3.6 所示, 如果用可分解的产生式系统来求解这个问题, 即把初始数据库分解为 3 个独立的子库 (C), (B) 和 (Z), 单独对它们使用规则, 直至全是 M 为止, 那么求解过程会简单得多, 如图 3.7 所示。它形成的是一个典型的与/或图, 其中的任一个解树都是产生式系统的解。

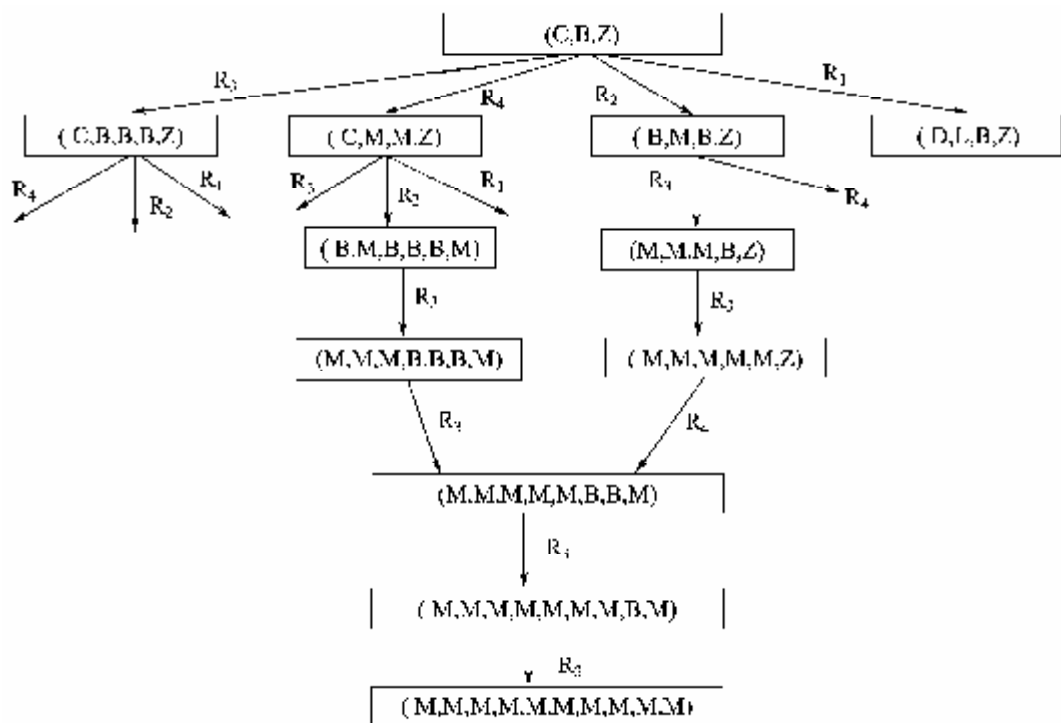


图 3.6 用图搜索产生式系统求解

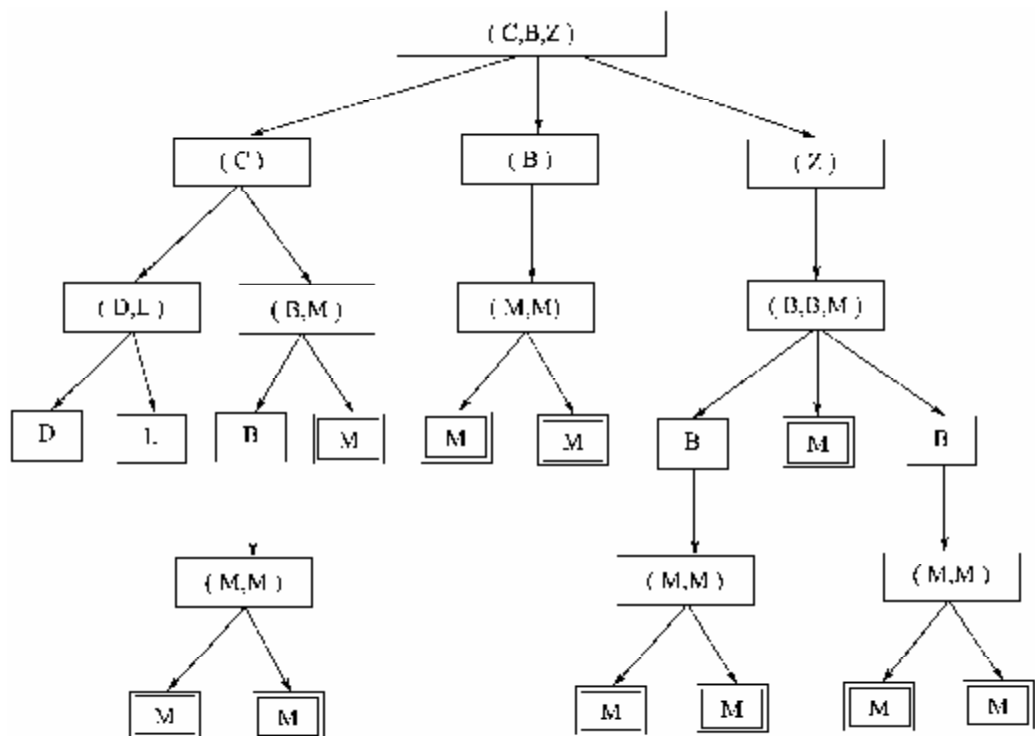


图 3.7 可用分解的产生式系统求解

3.4 专家系统

专家系统的研究致力于在具体的专门领域内建立高性能的程序，其实质就是把与领域问题求解相关的知识有机地结合到程序设计之中，使程序能够像人类专家一样进行推理、学习、解释，从而实现问题的求解。专家系统的内核通常是为一一定类型的知识表示，如规则、逻辑等而设计的，并且专家系统中知识表示的方式也直接影响着专家系统的开发、效率、速度及其维护。

专家系统是人工智能的一个重要分支。自 1968 年 Feigenbaum 等人研制成功第 1 个专家系统 DENDRAL 以来，专家系统技术已经获得了迅速发展，广泛地应用于医疗诊断、图像处理、石油化工、地质勘探、金融决策、实时监控、分子遗传工程、教学和军事等多个领域中，促进了人工智能基本理论和基本技术的研究与发展。目前，它已成为人工智能中一个最活跃、最有成效的研究领域。

3.4.1 专家系统的概念与组成

对于专家系统，至今没有一个确切的定义，不过总结各种陈述可以得出，专家系统就是一种在相关领域中具有专家水平解题能力的、包含知识和推理的智能程序系统。这种程序与传统的“应用程序”有本质的区别。在专家系统中，求解问题的知识已不再隐含在程序和数据结构之中，而是单独构成一个知识库，即传统的“数据结构+算法=程序”的应用程序模式发生了变化，使之成为“知识+推理=系统”的模式。它能运用领域专家多年积累的经验与专门知识，模拟人类专家的思维过程，求解需要专家才能解决的困难问题。其一般结构如图 3.8 所示。

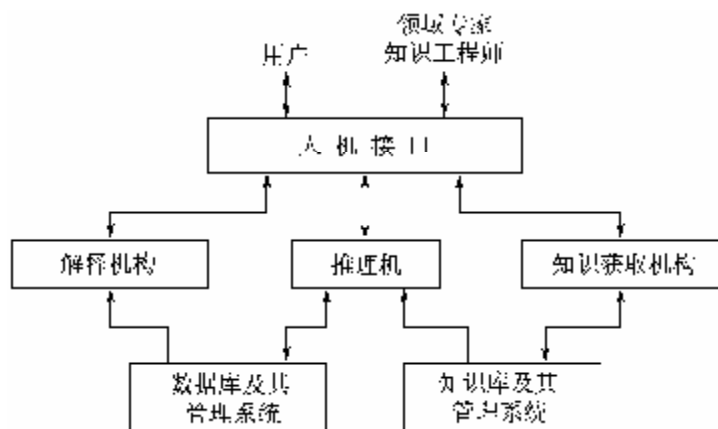


图 3.8 专家系统的一般结构

1. 知识获取机构

很多资料上都称知识获取是构造 ES 的“瓶颈”。实际上，它是成功构造专家系统中非

常重要的、也是非常困难的一部分，是 ES 研究的关键。它的任务是把专家对书本上的知识、客观世界的认识和理解进行选择、抽取、汇集、分类和组织，将它们转化为计算机可以利用的形式。对于大的复杂系统，要很好地完成这一任务非常困难。

2. 知识库及其管理系统

知识库主要用来存储某领域专家系统的专门知识，为了建立知识库，要解决知识获取和知识表示问题。知识获取涉及知识工程师如何从专家那里获得专门知识的问题；知识表示则要解决如何用计算机能够理解的形式表达并存储知识的问题。比如，对于常用的知识表示方法——产生式系统，知识库包含“事实”和“规则”。“事实”是短期信息(short term)，它可以在系统运行中不断改变，相对而言，“规则”是有关怎样生成新的事实以及如何根据现有事实得出假设的长期信息(long term)。

3. 数据库及其管理系统

这里的数据库用于存储领域或问题的初始数据和推理过程中得到的中间数据(信息)，即被处理对象的一些当前事实。数据库又称为“黑板”，它是由数据库管理系统进行管理的，这与一般程序设计中的数据库管理没有什么区别，只是应使数据的表示方法与知识的表示方法保持一致。

需注意的是，知识库与传统的数据库不一样：数据库一般是被动的，知识库则更有创造性；数据库中的事实是固定的，而知识库总是不断补充新的知识。

4. 推理机

推理机是专家系统的“思维”机构，是构成专家系统的核心部分。它的功能是根据一定的推理策略从知识库中选取有关知识，对用户提供的证据进行推理，直到得出相应的结论为止。推理机包括推理方法和控制策略两部分。

推理方法：推理分精确与不精确两种。

精确推理指把领域知识表示成必然的因果关系，推理的结论或是肯定的，或是否定的。

不精确推理指在“公理”（如领域专家给出的规则强度和用户给出的原始证据的不确定性）的基础上，定义一组函数，求出“定理”（非原始证据的命题）的不确定性度量。

专家系统中主要使用不精确推理，在这种推理中根据的事实可能是不充分的，依据的知识可能是不完整的经验性知识，这导致了这种推理要比精确推理复杂得多。

常用的不精确推理模型如下：

- (1) 确定性理论；
- (2) 主观 Bayes 方法；
- (3) 可能性理论；
- (4) 证据理论；
- (5) 模糊逻辑。

这些方法的基本思想是给各个不确定性的知识以某种确定性因子。在推理过程中，依某种算法计算各中间结果的确定因子，再沿着推理链传播这种不确定性，直到到达结论。当结论的确定性因子超过某个阈值后，结论便可成立。

此外，在确定性推理中，一个规则被激活的条件是它的前提为真，而在事实不充分的情况下，是否可激活某条规则，要视与规则前提所匹配的事实为真的程度而定。

这些处理方法是把人类那种不确定的“预感性”的推理定量化，变成了数字计算机可以解决的问题，这使许多专家系统确实具有了相当于人类专家水平的问题求解能力。

控制策略：控制策略主要指推理方向的控制及推理规则的选择策略。

推理有正向推理、反向推理及正反向混合推理。

正向推理由原始数据出发向结论方向推理，即所谓的事实驱动方式。其推理过程为：系统根据用户提供的原始信息，在知识库中寻找能与之匹配的规则，若找到，则将该知识块的结论部分作为中间结果，利用这个中间结果继续与知识库中的规则匹配，直到得出最终结论。

单纯的正向推理简单、易实现，但目的性不强，需要用启发性知识控制中间结论的选取，其中包括必要的回溯。另外，由于不能反推，系统的解释功能要受到影响。

反向推理先提出假设，然后由此出发，进一步寻找支持假设的证据，即所谓的目标驱动方式。当所需的证据与用户提供的原始信息相匹配时，推理成功。

显然，这些推理在选择目标时具有很大的盲目性。因此，反向推理比较适合结论单一或直接提出结论要证实的系统。

正反向混合推理，先根据原始数据通过正向推理帮助提出假设，再用反向推理进一步寻找支持假设的证据，反复这个过程。

正反向混合推理集中了正向与反向推理的优点，但其控制策略较前两者复杂。

推理机的性能与构造一般与知识的表示方式及组织方式有关，但与知识的内容无关，这有利于保证推理机与知识库的相对独立性，当知识库中的知识有变化时，无须修改推理机。但是，如果推理机的搜索策略完全与领域问题无关，那么它将是低效的，当问题规模较大时，这个问题就更加突出。为了解决这个问题，目前专家系统一方面为了提高系统的运行效率而使用了一些与领域有关的启发性知识，另一方面又为了保证推理机与知识库的相对独立性而采取了用元知识来表示启发性知识的方法。

5. 解释器

解释器能够对自己的行为做出解释，回答用户提出的“为什么？”、“结论是如何得出的？”等问题，是专家系统区别于一般程序的重要特征之一，也是它取信于用户的一个重要措施。另外，通过对自身行为的解释还可帮助系统建造者发现知识库及推理机中的错误，有助于对系统的调试及维护。因此，无论是对用户还是对系统自身，解释器都是不可缺少的。

解释器由一组程序组成，它能跟踪并记录推理过程，当用户提出询问需要给出解释时它将根据问题的要求分别做相应的处理，最后把解答用约定的形式通过人机接口输出给用户。

6. 接口

接口又称人机界面，主要用于系统与用户之间的双向信息交换，它包括以下几方面：

(1) 能够理解人按某种规定语言输入的事实和询问。大多数系统自己规定某种专用语

言供用户使用，系统中有一个语言翻译器，它对接收的语言进行语法分析和语义理解，然后转换成内部表示形式。将来的发展要求系统能够直接接受自然语言，或能听懂人的话，这就要求系统具有自然语言理解和语音识别等功能。

(2) 系统能向人提问。系统不仅能根据需要在运行中向用户索取事实和证据，而且能在发现故障（如知识库中有错误信息）时向专家请教，这相当于系统在向外界学习。

(3) 能够解释自己的工作过程和结论。

(4) 系统能回答提问。系统中事先存有一些专业知识可供用户提问，并回答用户的提问。将来，专家系统要能够在各个领域成为得心应手的咨询工具，甚至直接参与决策，因此，除了增加它的知识的深度和广度、提高系统的可靠性外，还应该使系统能够模拟智能活动的全过程。

3.4.2 专家系统的类型

按照专家系统所求解问题的性质，可把它分为下列几种类型。

1. 解释专家系统

解释专家系统的任务是通过已知信息和数据的分析与解释，确定它们的含义。

这样的专家系统具有下列特点：

- 系统处理的数据量很大，而且这些数据往往是不准确的、有错误的或不完全的。
- 系统能够从不完全的信息中得出解释，并能对数据做出某些假设。

解释专家系统的例子有染色体分类、PROSPECTOR 地质勘探数据解释和丘陵找水等实用系统。

2. 预测专家系统

预测专家系统的任务是通过过去和现在已知状况的分析，推断未来可能发生的动作。预测专家系统具有下列特点：

- 系统处理的数据随时间变化，而且可能是不准确和不完全的。
- 系统需要有适应时间变化的动态模型，能够从不完全和不准确的信息中得出预报，并达到快速响应的要求。

预测专家系统的例子有气象预报、军事预测、人口预测、交通预测、经济预测等。例如，恶劣气候(包括暴雨、飓风、冰雹等)预报、战场前景预测和农作物虫害预报等专家系统。

3. 诊断专家系统

诊断专家系统的任务是根据观察到的情况(数据)来推断出某个对象机能失常(即故障)的原因。诊断专家系统具有下列特点：

- 能够了解被诊断对象或客体各组成部分的特性以及它们之间的联系。
- 能够区分一种现象及其所掩盖的另一种现象。
- 能够向用户提出测量的数据，并从不确定信息中得出尽可能正确的诊断。

诊断专家系统的例子特别多，有医疗诊断，电子机械和软件故障诊断以及材料失效诊断等。用于抗生素治疗的 MYCIN、肝功能检验的 PUFF、青光眼治疗的 CASNET、内科的 INTERNIST-I 和血清蛋白诊断等医疗诊断专家系统，IBM 公司的计算机故障诊断，火电厂锅炉给水系统故障检测与诊断系统，雷达故障诊断系统等都是国内外颇有名气的实例。

4. 设计专家系统

设计专家系统的任务是根据设计要求，求出满足设计问题约束的目标配置。设计专家系统具有下列特点：

- 善于从多方面的约束中得到符合要求的设计结果。
- 系统需要检索较大的可能解空间。
- 善于分析各种种子问题，并处理好子问题间的相互作用。
- 能够试验性地构造出可能设计，并易于对所得设计方案进行修改。
- 能够使用已被证明是正确的设计来解释当前的(新的)设计。

设计专家系统涉及电路(如数字电路和集成电路)设计、土木建筑工程设计、机械产品设计和生产工艺设计等。比较有影响的专家设计系统有浙江大学的花布立体感图案设计和花布印染专家系统、大规模集成电路设计专家系统等。

5. 规划专家系统

规划专家系统的任务在于寻找出某个能够达到给定目标的动作序列或步骤。规划专家系统的特点如下：

- 所要规划的目标可能是动态的或静态的，因而需要对未来动作做出预测。
- 所涉及的问题可能很复杂，要求系统能抓住重点，处理好各子目标间的关系和不确定的数据信息，并通过试验性动作得出可行规划。

规划专家系统可用于机器人规划、交通运输调度、工程项目论证、通信与军事指挥以及优良作物施肥方案规划等。比较典型的规划专家系统的例子有军事指挥调度系统、ROPES 机器人规划专家系统、汽车和火车运行调度专家系统以及小麦和水稻施肥专家系统。

6. 监视专家系统

监视专家系统的任务在于对系统、对象或过程的行为进行不断观察，并把观察到的行为与其应当具有的行为进行比较，以发现异常情况，发出警报。监视专家系统具有下列特点：

- 系统应具有快速反应能力，在造成事故之前及时发出警报。
- 系统发出的警报要有很高的准确性。在需要发出警报时发警报，在不需要发出警报时不得轻易发警报(假警报)。
- 系统能够随时间和条件的变化而动态地处理其输入信息。

监视专家系统可用于核电站的安全监视、防空监视与警报、国家财政的监控、传染病疫情监视及农作物病虫害监视与警报等。粘虫测报专家系统是监视专家系统的一个实例。

7. 控制专家系统

控制专家系统的任务是自适应地管理一个受控对象或客体的全面行为，使之满足预期的要求。控制专家系统的特点是：能够解释当前情况，预测未来可能发生的情况，诊断可能发生的问题及其原因，不断修正计划，并控制计划的执行。也就是说，控制专家系统具有解释、预报、诊断、规划和执行等多种功能。

空中交通管制、商业管理、自主机器人控制、作战管理、生产过程控制和生产质量控制等都是控制专家系统的潜在应用方面。

8. 调试专家系统

调试专家系统的任务是对失灵的对象给出处理意见和方法。

调试专家系统的特点是同时具有规划、设计、预报和诊断等专家系统的功能。

调试专家系统可用于新产品或新系统的调试，也可用于维修站进行被修设备的调整、测量与试验。在这方面的实例还很少见。

9. 教学专家系统

教学专家系统的任务是根据学生的特点、弱点和基础知识，以最适当的教案和教学方法对学生进行教学和辅导。

教学专家系统的特点如下：

- 同时具有诊断和调试等功能。
- 具有良好的人机界面。

已经开发和应用的教学专家系统有美国麻省理工学院的 MACSYMA 符号积分系统，中国一些大学开发的计算机程序设计语言和物理智能计算机辅助教学系统以及机器人语言训练专家系统等。

10. 修理专家系统

修理专家系统的任务是对发生故障的对象(系统或设备)进行处理，使其恢复正常。修理专家系统具有诊断、调试、计划和执行等功能。

美国贝尔实验室的 ACI 电话和有线电视维护修理系统是修理专家系统的一个应用实例。此外，还有决策专家系统和咨询专家系统等。

3.4.3 专家系统的特点

1. 专家系统的共同特点

已经介绍过各类专家系统的特点。在总体上，专家系统还具有下列 3 个共同的特点：

- 启发性 专家系统能运用专家的知识与经验进行推理、判断和决策。世界上大部分工作和知识都是非数学性的，只有一小部分人类活动是以数学公式为核心的。即使是化学和物理学科，大部分也是靠推理进行思考的。对于生物学、大部分医

学等情况也是这样。企业管理的思考几乎全靠符号推理，而不是数值计算。

- 透明性 专家系统能够解释本身的推理过程和回答用户提出的问题，以便让用户能够了解推理过程，提高对专家系统的信赖感。例如，一个医疗诊断专家系统诊断某病人为肺炎，而且必须用某种抗生素治疗，那么这一专家系统将会向病人解释为什么确诊他患肺炎，而且必须用某种抗生素治疗，就像一位医疗专家对病人详细解释病情一样。
- 灵活性 专家系统能不断地增长知识，修改原有知识，不断更新。由于这一特点，使得专家系统具有十分广泛的应用领域。

2. 专家系统的优点

近十多年来，专家系统获得迅速发展，应用领域越来越广，解决实际问题的能力越来越强。这是专家系统的优良性能以及对国民经济的重大作用决定的。具体地说，包括下列几个优点：

- 专家系统能够高效率、准确、周到、迅速和不知疲倦地进行工作。
- 专家系统解决实际问题时不受周围环境的影响，也不可能遗漏忘记。
- 可以使专家的专长不受时间和空间的限制，以便推广珍贵和稀缺的专家知识。
- 专家系统能促进各领域的发展，它使各领域专家的专业知识和经验得到总结并精炼，能够广泛有力地传播专家的知识、经验和能力。
- 专家系统能汇集多领域专家的知识 and 经验以及他们协作解决重大问题的能力，所以它拥有更渊博的知识、更丰富的经验和更强的工作能力。
- 专家系统的研制和应用，具有巨大的经济效益和社会效益。
- 军事专家系统的水平是一个国家国防现代化的重要标志之一。
- 研究专家系统能够促进整个科学技术的发展。专家系统对人工智能的各个领域的发展起了很大的促进作用，并将对科技、经济、国防、教育、社会和人民生活产生极其深远的影响。

3.4.4 专家系统的开发工具

由于专家系统具有十分广泛的应用领域，而每个系统一般只具有某个领域专家的知识，如果在建造每个具体的专家系统时，一切都从头开始，就必然会降低工作效率。人们已经研制出一些比较通用的工具，作为设计和开发专家系统的辅助手段和环境，以求提高专家系统的开发效率、质量和自动化水平。这种开发工具或环境，就称为专家系统开发工具。

现有的专家系统开发工具，大致分为下面几类：

(1) 面向 AI 的通用程序设计语言，如 Prolog, LISP, CLISP 等。由于它们可以将符号直接写在程序中，因此能够以接近自然语言的方式表达知识和规则以及推理过程。这些语言还可以直接生成新知识，因而在建立专家系统时特别有效。

(2) 通用知识表示语言。这是针对知识工程发展起来的程序设计语言，这些语言并不与具体的体系和范例有紧密联系，也不局限于实现任一特殊的控制策略，因而便于实现较

广泛的问题。

由于不同的应用目的知识类型不同，其知识表达方式也不同，所以开发了若干流行的知识表示语言。如产生式语言系统 OPS-5，基于框架理论的知识表示语言 PLL、KRL、UNITS，还有 LOOPS，它集中了 4 种编程方式，即面向目标、面向数据、面向规则和它们组合。在面向过程的语言 INTERLISP-D 程序设计环境下，允许设计者选择最适合其目的的那种方式。LOOPS 还包括用于创建和调整知识库系统的编程环境。

(3) 专家系统的外壳，有的称为骨架。这些系统通常提供知识获取模块、推理机制、解释功能等，只要加上领域专门知识就可以构成一个专家系统。这些骨架有的是从原有的专家系统中演变过来的，因此它们的控制策略局限于原有系统提供的一些控制策略。这类系统典型的代表有 EMYCIN，KAS 和 EXPERT 等。

(4) 通用化专家系统构造工具，也称组合式专家系统研制工具。它向用户提供多种知识表示方法和多个推理控制机构，使用户可以选择和设计各种组成部件，非常方便地组合，设计自己所需的专家系统，该系统的典型代表是 AGE 等。

3.4.5 新一代专家系统研究

自从世界上第 1 个专家系统 DENDRAL 问世以来，专家系统已经走过了 30 余年的发展历程。从技术角度看，基于知识库（特别是规则库）的传统专家系统已趋于成熟，但仍存在不少问题，诸如知识获取问题、知识的深化问题、不确定性推理问题、系统的优化和发展问题、人机界面问题、同其他应用系统的融合与接口问题等，都还未得到满意解决。为此，人们就针对这些问题，对专家系统做进一步研究，引入了多种新思想、新技术，提出了形形色色的所谓新一代专家系统。

1. 新一代专家系统的特征

新一代专家系统应具有以下特征：

- (1) 并行分布式处理；
- (2) 多专家协同工作；
- (3) 高级语言和知识语言描述；
- (4) 具有学习功能；
- (5) 引入新的推理机制；
- (6) 具有纠错和自完善能力；
- (7) 先进的智能人机接口。

如果要求一个专家系统完全实现上述特征，将是一个相当困难的任务。因此，一个新一代专家系统往往只在单项或几项指标上满足上述特征要求。

2. 分布式专家系统

这种专家系统具有分布处理的特征，其目的在于把一个专家系统的功能经分解以后分布到各个处理机上去并行工作，从而在总体上提高系统的处理效率。为设计一个分布式专家系统，一般需要解决下述问题。

(1) 功能分布：把系统功能分解为多个子功能，并均衡地分配到各个处理结点上。每个结点上实现一个或两个子功能，各结点合在一起作为一个整体完成一个完整的任务。

(2) 知识分布：根据功能分布的情况，把有关知识合理划分后分配到各个处理结点上。

(3) 接口设计：各个部分之间要相互独立，接口要易于通信、易于同步。

(4) 系统结构：系统结构一方面与问题本身的性质有关，另一方面与硬件环境有关。

(5) 驱动方式：系统各模块之间的驱动方式有以下几种，控制驱动，即当需要某个模块工作时，就直接将控制转到该模块，或将它作为一个过程直接进行调用；数据驱动，即当一个模块的输入数据齐备后，该模块就自动启动工作；要求驱动，也称为目的驱动，即从最顶层的目标开始逐层驱动下层的子目标；事件驱动，即当且仅当一个模块的相应事件集合中的所有事件都已经发生时，才驱动该模块开始工作。

3. 协同式专家系统

当前现存的专家系统一般为单个专家的系统，其解决问题的领域很窄，很难获得满意的应用。协同式专家系统是克服单专家系统局限性的一个重要途径。协同式专家系统也称为“群专家系统”，是一种能综合若干个相近领域或一个领域的多个方面的分专家系统相互协作，共同解决一个更广领域问题的专家系统。

协同式专家系统和分布式专家系统有一定的共性，它们都会涉及多个分专家系统。但是，分布式强调的是处理的分布和知识的分布，它要求系统必须在多个处理机上运行；协调式强调的是分系统之间的协同合作，各分专家系统也可以在同一个处理机上运行。要设计协同式专家系统，一般需要解决以下问题。

(1) 任务的分解：根据领域知识，将确定的总任务合理地划分为若干个子任务（各个子任务间允许有一定的重叠），每个子任务对应着一个分专家系统。

(2) 公共知识的导出：把各子任务所需知识的公共部分分离出来形成一个公共知识库，供各分专家系统共享。

(3) “讨论”方式：用“黑板”（即设在内存的一个可供各分专家系统随机存取的存储区）作为各分专家系统进行讨论的园地。

(4) 裁决问题：所谓裁决问题是指如何由多个分专家系统来决定某个问题。其解决办法与问题的性质有关，若为选择问题，可采用少数服从多数的方法；若为评分问题，则可采用加权平均法等办法；若为互补问题，则可采用互相配合的方法。

(5) 驱动方式：这个问题与分布式专家系统中所采用的驱动方式基本上是一样的。在分布式专家系统中介绍的驱动方式对协同式专家系统同样适用。

4. 深层知识专家系统

深层知识专家系统不仅具有专家经验性表层知识，而且具有深层次的专业知识。这样，系统的智能就更强了，也更接近于专家水平了。例如，一个故障诊断专家系统，如果不仅有专家的经验知识，而且也有设备本身的原理性知识，那么对于故障判断的准确性将会进一步提高。要做到这一点，这里存在一个如何把专家知识与领域知识融合的问题。

5. 模糊专家系统

主要特点是通过模糊推理解决问题。这种系统善于解决那些含有模糊性数据、信息或知识的复杂问题，但也可以通过把精确数据或信息模糊化，然后通过模糊推理进行处理解决复杂问题。

这里所说的模糊推理包括基于模糊规则的串行演绎推理和基于模糊集并行计算（即模糊关系合成）的推理。对于后一种模糊推理，其模糊关系矩阵也就相当于通常的知识库，模糊矩阵的运算方法也就相当于通常的推理机。

6. 神经网络专家系统

利用神经网络的自学习、自适应、分布存储、联想记忆、并行处理，以及鲁棒性和容错性强等一系列特点，用神经网络来实现专家系统的功能模块。

神经网络专家系统的建造过程是：先根据问题的规模，构造一个神经网络，再用专家提供的典型样本规则，对网络进行训练，然后利用学成的网络，对输入数据进行处理，便得到所期望的输出。

可以看出，这种系统把知识库融入网络之中，而推理过程就是沿着网络的计算过程。基于神经网络的这种推理，实际是一种并行推理。

这种系统实际上是自学习的，它将知识获取和知识利用融为一体，而且它所获得的知识往往还高于专家知识，因为它所获得的知识是从专家提供的特殊知识中归纳出的一般知识。

这种专家系统还有一个重要特点，那就是它具有很好的鲁棒性和容错性。

研究发现，模糊技术与神经网络存在某种等价和互补关系。于是，人们就将二者结合起来，构造模糊—神经网络或神经—模糊系统，从而开辟了将模糊技术与神经网络技术相结合、将模糊系统与神经网络系统相融合的新方向。

本章小结

本章介绍人工智能编程的基础知识，内容包括命题逻辑、一阶谓词逻辑、产生式系统、专家系统等。

命题逻辑，主要介绍了命题的概念、命题联接词、合式公式、真值指派、等价与蕴含、命题定律、析取范式与合取范式、命题逻辑的推论规则以及命题逻辑的局限性。

一阶谓词逻辑，主要介绍了谓词与量词的概念、谓词逻辑的合式公式、谓词公式中的自由变元与约束变元、谓词公式的解释、含有量词的等价式和蕴含式、谓词逻辑中的推论规则、谓词公式的范式与斯柯林标准形。

产生式系统，主要介绍了产生式系统的基本组成、基本类型、用产生式系统进行问题建模的方法及求解问题的过程，并针对八数码游戏问题，详细介绍了不可回撤的搜索策略与回溯控制方法，最后介绍了两种典型的系统，即回溯式产生式系统和可分解的产生式系统。

专家系统，主要介绍了专家系统的概念与组成、专家系统的类型、专家系统的一般特点、专家系统开发工具以及对新一代专家系统的展望。

习题 3

1. 构造下列公式的真值表，指出其中的永真式、永假式。

(1) $Q \wedge (P \rightarrow Q) \rightarrow P$

(2) $\sim (P \vee (Q \wedge R)) \leftrightarrow (P \vee Q) \wedge (P \vee R)$

2. 用命题定律和谓词定律证明下列等价式。

(1) $P \rightarrow (Q \rightarrow P) \leftrightarrow \sim P \rightarrow (P \rightarrow Q)$

(2) $\sim (P \leftrightarrow Q) \leftrightarrow (P \vee Q) \wedge \sim (P \wedge Q)$

(3) $(\forall x)(P(x) \vee Q(x)), (\forall x) \sim P(x) \Rightarrow (\forall x) Q(x)$

(4) $(\exists x) P(x) \rightarrow (\forall x) Q(x) \Rightarrow (\forall x)(P(x) \rightarrow Q(x))$

3. 求下列公式的主析取范式和主合取范式。

(1) $P \wedge (Q \vee \sim R) \vee (\sim Q \wedge R)$

(2) $\sim ((P \wedge Q) \vee R) \wedge (P \vee (Q \wedge R))$

4. 已知公理集为 P , $(P \wedge Q) \rightarrow R$, $(S \vee T) \rightarrow Q$, T , 求证 R 。

5. 化下列公式成子句形式。

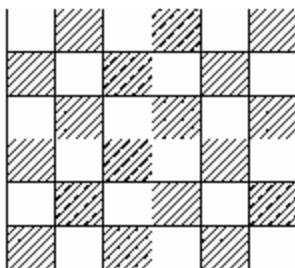
(1) $\{ \sim [(\forall x)P(x)] \} \rightarrow (\exists x)[\sim P(x)]$

(2) $\sim (\forall x)\{P(x) \rightarrow \{(\forall y)[P(y) \rightarrow P(f(x, y))]\} \wedge \sim (\forall y)[Q(x, y) \rightarrow P(y)]\}$

6. 对猴子摘香蕉问题，给出产生式系统描述。

一个房间里，天花板上挂有一串香蕉，有一只猴子可在房间里任意活动（到处走动、推移箱子、攀登箱子等）。设房间里还有一只可被猴子移动的箱子，且猴子登上箱子时才能摘到香蕉，问猴子在某一状态下（设猴子位置为 a ，箱子位置为 b ，香蕉位置为 c ），如何行动可摘取到香蕉。

7. 用产生式系统求解六皇后问题：在 6×6 的方格棋盘上，放置 6 个皇后，使他们中间的任何两个都不在同一行、同一列及同一对角线上。



8. 什么是专家系统？它有哪些基本特征？

9. 专家系统有几种分类方法？它们都可以分为哪几种主要类型？

10. 专家系统有哪些基本部分？每一部分的主要功能是什么？

11. 知识获取的主要任务是什么？为什么说它是专家系统建造的“瓶颈”问题？
12. 专家系统建造的原则是什么？建造一个专家系统要经历哪几个阶段？
13. 有哪几类专家系统开发工具？各有什么特点？
14. 按下列规则，写出一个分类专家系统：
 - (1) 有毛的动物是哺乳类；
 - (2) 有奶的动物是哺乳类；
 - (3) 有羽毛的动物是鸟类；
 - (4) 若动物会飞且会生蛋，则它是鸟类；
 - (5) 吃肉的哺乳类是肉食动物；
 - (6) 犬牙利爪、眼睛向前的是肉食动物；
 - (7) 反刍食物的哺乳类动物属偶蹄类；
 - (8) 有蹄的哺乳类动物属蹄类；
 - (9) 黄褐色、有暗斑点的肉食类动物是金钱豹；
 - (10) 黄褐色、有黑条纹的肉食类动物是老虎；
 - (11) 长腿、长脖子、有黄褐色暗斑的有蹄类动物是长颈鹿；
 - (12) 有黑白条纹的有蹄类动物是斑马；
 - (13) 不会飞、长腿、长脖的鸟是鸵鸟；
 - (14) 不会飞、善游泳、黑白色的鸟是企鹅；
 - (15) 善飞的鸟是信天翁。

第 2 部分 编程指南

第 4 章 Visual Prolog 概述

第 5 章 Prolog 基础

第 6 章 类与对象

第 7 章 Visual Prolog 编程

第 8 章 编写 CGI 程序

第 9 章 编码风格

第 4 章 Visual Prolog 概述

本章首先介绍 Visual Prolog 6 的基本特性。然后通过实例，较为详细地介绍 Visual Prolog 6 的可视化开发环境（VDE），包括创建项目、建立项目、浏览项目、开发项目、调试项目等。

4.1 Visual Prolog 6 概述

Visual Prolog 6 是最新一代的 Visual Prolog 逻辑程序设计语言，是 Visual Prolog 5 和 PDC、Turbo Prolog 的后继产品。Visual Prolog 6 的目标是支持企业级的强调问题求解的复杂知识的程序设计。Visual Prolog 6 的发布是 PDC 历时 3 年开发的结果。经过持续不断的努力和对用户需求的深入考查，Visual Prolog 已经增加了如下功能：

- 一个独特的对象系统
- 多线程机制
- unicode 支持
- 改进的 DLL 支持
- 改进的函数支持
- 改进的异常处理
- 其他更多功能等

今天，Visual Prolog 6 是一个功能非常强大的、非常安全的程序设计语言，它以一致和一流的方式将许多编程范例结合在一起。Visual Prolog 是一个完备的程序设计环境，它提供如下设施：

- 图形开发环境
- 编译器
- 链接器
- 调试器

开发环境已经得到极大的改善，从而使编写程序更加简单，对高级任务可提供更好的帮助。它支持先进的客户/服务器和多级解决方案。使用 Visual Prolog，人们就能在 Microsoft Windows 平台建立企业级的应用程序。Visual Prolog 特别适用于处理复杂的知识问题。PDC 已经通过实例证明了这一点，它提供一些成功的应用系统案例如下：

- 职员计划
- 医院预约登记
- 机场决策支持
- 航班决策支持

- 商店室内调度

上述所有这些系统全部是用 Visual Prolog 写成的。

通过使用功能强大的对象系统，人们能够非常迅速地构筑一个应用，同时受益于非常宽松的耦合环境。这将能够缩短开发周期，减少维护费用。

PDC 提供免费的非商业的个人版本，从而为学习使用这个卓越的系统提供了良机。

4.2 Visual Prolog 6 基本特性

Visual Prolog 6 是最新一代的 Visual Prolog 逻辑程序设计语言，它可以创建 Win 32 平台的企业级应用程序。

Visual Prolog 6 是基于 Prolog 的强类型的面向对象程序设计语言。下面从语言特性、图形化开发环境、编译器、链接器、调试器等方面简要予以介绍。

4.2.1 语言特性

Visual Prolog 6 语言的主要特性如下：

- 基于 Horn 子句的逻辑程序设计语言。
- 完全面向对象。
- 对象谓词值（委派）。
- 强类型。
- 代数数据类型。
- 模式匹配与合一。
- 受控的不确定性机制。
- 完全集成的事实数据库。
- 自动的内存管理。
- 支持与 C/C++ 的直接链接。
- 支持对 Win32 API 函数的直接访问。

对象机制实现了系统和用户之间的松散耦合。对象只能通过接口来访问，接口与实现之间不过是松散耦合。类可以通过继承（或不继承）其他类来实现接口。

强大的类型检测、无需指针算法和自动内存管理的结合真正地避免了非法访问。

无非法访问一直是 Visual Prolog 的一个优势。正如 PDC 的一位用户所说：“对于 Visual Prolog，那种错误不存在”。Visual Prolog 6 毫无例外地继续保持这个优势。PDC 的目标是：避免在必须调用外部代码或建立指针算法时引起非法访问。

用非决定性搜索将符号数据类型、事实数据库和模式匹配结合起来，这样使得 Visual Prolog 非常适合处理复杂的结构化知识。

除了谓词值和对象，所有的 Visual Prolog 数据都有一个人性化的可读文本表示，它可以被写入并返回到程序。

4.2.2 图形化开发环境

Visual Prolog 集成开发环境可以更方便快捷地建立、测试和修改 Visual Prolog 应用程序。它在开发大型项目时非常有用。

- 项目窗口中的模块、包括文件和资源的树型结构，有助于将项目打包，从而给出了一个额外级别的抽象。
- 文本编辑器可以方便地进行文本编辑，浏览那些声明和实现。
- 对话框编辑器为设计对话框提供了标准控件。
- 菜单编辑器允许创建下拉式菜单和弹出式菜单。
- 工具栏编辑器允许创建各种工具栏。
- 图形编辑器可以方便地创建、查看和编辑图标、指针和小位图。
- 建造工具支持插入所需的程序包和包含指令。
- 浏览工具支持查找特定的实体，go to definition 和 go to declaration。

4.2.3 编译器

Visual Prolog 编译器是 20 世纪 80 年代的 Turbo Prolog 编译器的后继产品，Turbo Prolog 是第 1 个 Prolog 编译器。自那时开始，PDC 就一直开发并改进这个编译器，所以现在的 Visual Prolog 编译器是一个功能强大且高效的编译器，它可以用来：

- 为产生可单独执行的程序或 DLL 而创建目标文件。
- 解决声明间的交叉引用。
- 验证谓词模式。
- 执行强类型检查。
- 在构造器中验证事实初始化。
- 进行谓词分解。

4.2.4 链接器

Visual Prolog 拥有一个功能强大的链接器：

- 产生 EXE 可执行文件和 DLL 文件。
- 使用由最新的 Microsoft Visual C 编译器产生的 LIB 文件。

4.2.5 调试器

Visual Prolog 集成开发环境包含一个内建图形化调试器：

- 显示常见的调试器视图，如内存，堆栈，变量等。
- 显示类和对象事实的值。
- 进行单步跟进（trace into）、单步越进（step over）等。

- 包括额外的调试步骤：单步跳出、运行到 Prolog 代码等。
- Fail 和 Exit 可视化等。

4.3 创建项目

从这一节开始，将给出一个可视化开发环境（VDE）的综述。这个 VDE 是用来创建、开发和维护 Visual Prolog 项目的。也就是说，在一个项目文件中，将使用这个 VDE 来完成如下任务。

创建（creation）项目：即用 VDE 来创建一个项目。在项目创建期间，可以选择该项目的一些重要特性，如该项目是产生一个可执行文件还是产生一个 DLL，是使用 GUI 还是只使用文本方式等。

建立（building）项目：即建立一个项目，在 VDE 中进行编译和链接等。

浏览（browsing）项目：VDE 和编译器收集关于该项目的信息，这些信息以各种方式被用来进行实体的快速定位等。

开发（development）项目：在项目的开发和维护期间，VDE 被用来给项目添加源文件和 GUI 实体、删除源文件和 GUI 实体，以及编辑源文件和 GUI 实体。

调试（debugging）项目：VDE 还被用来调试项目。在程序运行期间，调试器可用来跟踪程序的执行，探索程序的状态。

本章最后将较为详细地回顾上述这些事件。值得注意的是，这里将首先创建一个项目，并将这个项目贯穿于全章。

首先创建一个项目。在菜单中选择 Project -> New，VDE 将对此作出响应，出现一个包含项目各种特性的对话框，如图 4.1 所示。

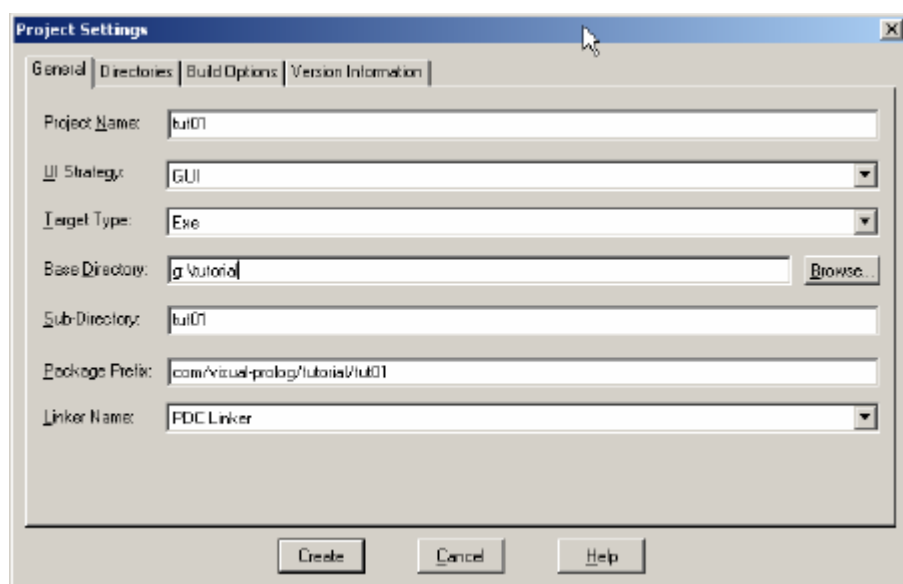


图 4.1 项目设置选项对话框

选择项目名为 `tut01`。项目名也作为将要产生的目标文件名。在此例中，目标文件是一个 `exe` 文件，故目标文件名将是 `tut01.exe`。选择 UI 策略为 `GUI`，即该程序是一个 GUI 程序，带有图形用户界面。

基本目录（base directory）是一切项目的“基地”。为此，可以选择一个方便的目录位置。新项目将在基本目录的一个子目录中进行创建，默认情况下，这个子目录名与项目名称同名。

在系统中创建项目时，常使用包前缀（package prefix）。关于包的概念，后面的章节还将详细解释。在此例中，源程序文件将与其他人共享前缀，如果这个前缀不与其他前缀相冲突，这是一个好的做法。此时，暂不需要考虑其余的选项。

现在，单击“创建”按钮，创建该项目。VDE 将显示出如图 4.2 所示的情形。

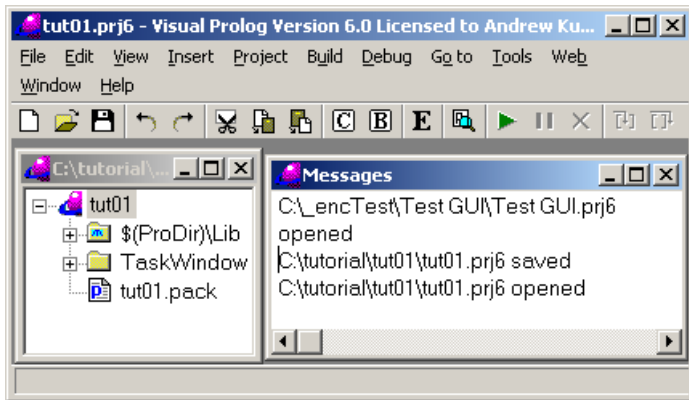


图 4.2 项目创建过程

左面的窗口是项目窗口（project window），它包含项目中有关实体的各种信息。这个窗口现在尚不包含大量的信息，但当编译该项目时，各种信息就会被添加进来。后面将会较详细地分析在编译该项目时这个窗口所包含的各种信息。

左面或底部是消息窗口（messages window），它将包含各种状态信息和进展信息。

4.4 建立项目

在做任何修改之前，首先要建立该项目，即编译和链接该项目。在“建立”菜单中，可以找到建立、编译和执行项目的菜单命令。

如果选择“执行”命令，则该项目首先将会执行建立操作，这取决于执行程序的版本日期。因此，可以直接选择“执行”命令（或直接按下 `F9` 键）。

如果还没有对 Visual Prolog 进行注册，将会出现一个专门的屏幕信息进行提醒。建议对 Visual Prolog 进行注册，也可以选择“继续”。

在消息窗口，VDE 显示哪些文件被编译等。

如果建立进程成功，就像此例一样，所创建的程序被执行。本例现在的结果是只可以看见一个小的什么也不做的 GUI 程序。值得注意的是，这个程序看起来有点像 VDE 本身。

因为 VDE 实际上就是一个 Visual Prolog 程序，所以这点并不是巧合。

在本章后面还将看到，如果在程序建立过程中编译器或链接器检测到错误，将会发生什么样的情形。

4.5 浏览项目

现在将目光转到项目窗口中的项目树（project tree），并对其稍加解释。项目树本身是以标准的窗口树控件进行显示的，读者对此用法已经比较熟悉。这里将集中精力在该树的内容上。项目树的结构和内容如图 4.3 所示。

图中顶部结点代表项目，其余结点是项目目录。

紧下面是逻辑结点\$(ProDir)，它表示 Visual Prolog 的安装目录。这个目录包含来自 Visual Prolog 系统的库和库代码。

接下来的目录是任务窗口（task window），它是该项目目录的一个子目录。这个目录包含产生任务窗口、菜单、工具栏及关于（about）对话框等所需要的全部代码。

最后可以看到若干文件。Visual Prolog 使用以下约定：

- *.ph 文件是程序包的头文件（package headers）。一个程序包是类和接口的一个集合，程序包常被当作一个积木块使用。
- *.pack 文件是程序包。它们包含相应的.ph 文件的实现或定义。
- *.i 文件包含一种接口（interface）。
- *.cl 文件包含一个类声明（class declaration）。
- *.pro 文件包含一个类实现（class implementation）。

如果完全展开 tut01.cl 结点，其情形如图 4.4 所示。

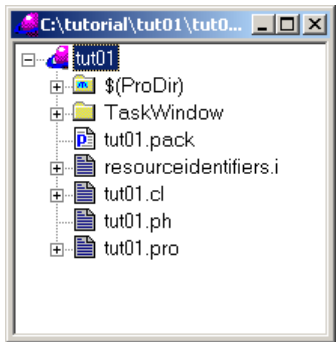


图 4.3 项目窗口中的项目树

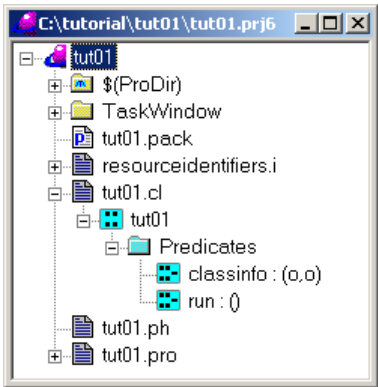


图 4.4 项目树展开的内容

这个子目录表明，文件 tut01.cl 包含一个叫做 tut01 的类，这个类又包含两个分别叫做 classinfo 和 run 的谓词。所谓“谓词”就是子例程，在本章中将不对它们做深层次的探究。

如果折叠起这个结点，重新展开任务窗口结点，可以看到如图 4.5 所示的目录树。

新出现的几个类型的结点的含义如下：

- *.dlg 文件包含一个对话框（dialog）；
- *.win 文件包含一个窗口（window）；
- *.mnu 文件包含一个菜单（menu）；
- *.cur 文件包含一个光标（cursor）；
- *.ico 文件包含一个图标（icon）。

继续考查后还可以发现：

- *.tb 文件包含工具栏（toolbars）；
- *.bmp 文件包含位图（bitmaps）；
- *.lib 文件包含库（libraries）。

如果右击一个结点，一个关联菜单将出现，菜单中包含对这个特定结点进行适当操作的有关命令。

如果双击一个结点，则相应的实体将调出对应的编辑器。所有源代码文件都将在文本编辑器中进行编辑，而窗口资源，如对话框和菜单等将在图形编辑器中进行编辑。后面将进一步考查图形编辑器和文本编辑器。

某些实体在项目树中出现两次，这是因为它们既有一个声明，又有一个定义或实现。例如，tut01 类中的谓词 run，如图 4.6 所示。

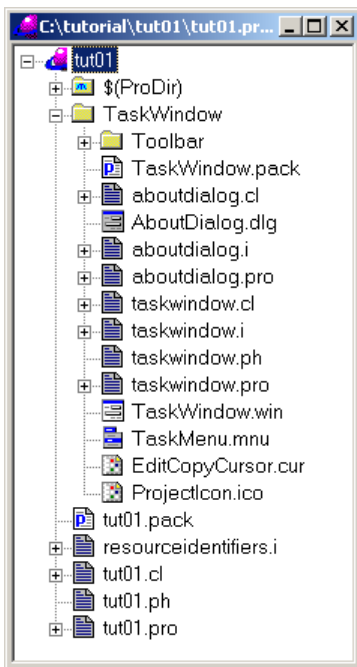


图 4.5 任务窗口项目树的内容

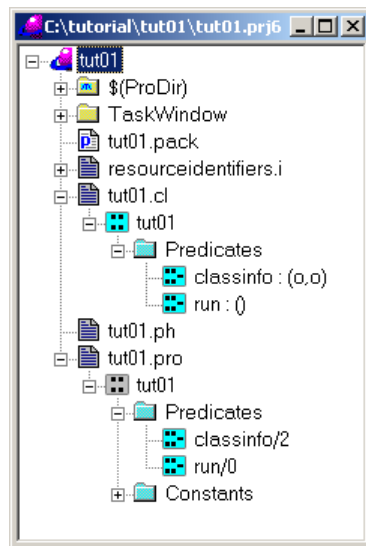


图 4.6 项目树展开后的内容

试着双击每个 run 结点，可以看到，显示 run 谓词的声明和定义的两个编辑器分别被打开。

VDE 还有其他的工具，以浏览指定的实体，但这些工具在此将不进行讨论。

4.6 开发项目

现在试着对该项目做修改。因为仍然未考虑如何用 Visual Prolog 进行编程，所以将使这些改变尽量简单一些。

这里故意引入一个错误，因而就可以看到错误窗口。首先，在文件 `tut01.pro` 中查找 `run` 谓词子句。如果在项目树中双击这两个 `run` 结点的后面的那个结点，文本编辑器将被打开，一个插字符号正好位于该结点处，如图 4.7 所示。

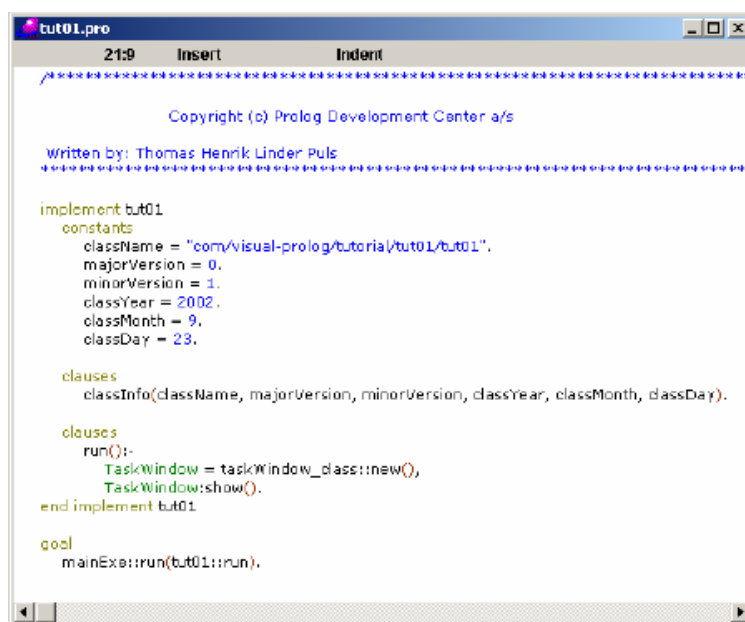


图 4.7 文本编辑器窗口

试着插入一个 `fail`，如下面的代码所示（注意 `show` 后面的逗号）。

```
clauses
  run() :-
    TaskWindow = taskWindow_class::new(),
    TaskWindow:show(),
    fail.
```

试图再次建立该项目，即直接按下 `F9` 键。系统将保存和编译该文件，但由于引入了一个错误，所以错误窗口被打开，如图 4.8 所示。

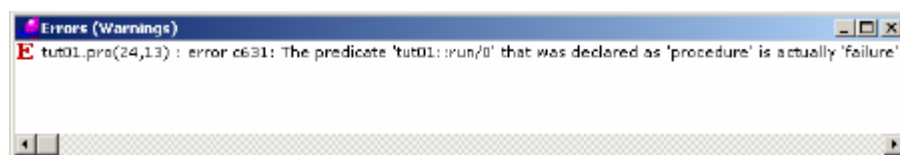


图 4.8 错误窗口

考虑错误信息的实际含义。一旦双击错误消息，就会发现，编辑器再次获得焦点，插入符号准确地指向刚刚插入的 `fail` 谓词。

去掉 `fail` 谓词，复原该代码，再次建立该项目。

接着，试图在关于对话框中进行修改。这并不是一个很明智的改变，但它却能说明一些问题。

首先，用对话框编辑器（dialog editor）打开 `about` 对话框。为此，在项目树中双击该对话框。必须双击的结点如图 4.9 所示。

一旦双击了这个结点，就会在对话框编辑器中看到该对话框和两个工具栏，如图 4.10 所示。

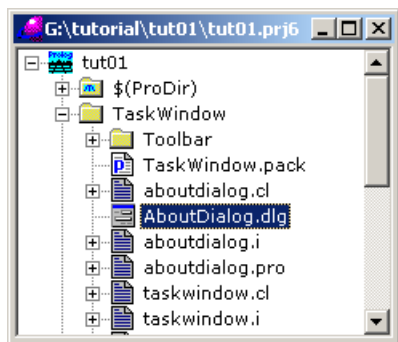


图 4.9 双击项目树中 `About` 对话框

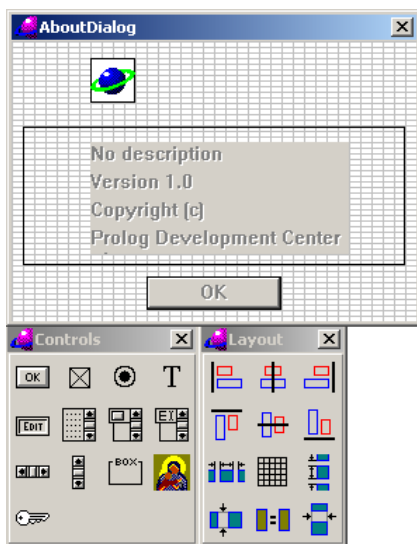


图 4.10 对话框编辑器

现在给这个对话框添加一个按钮。首先单击控件工具栏上的按钮图标，然后在 `AboutDialog` 对话框窗口中邻近项目图标的旁边单击该对话框。结果一个窗口弹出来，如图 4.11 所示。

改变这里的文本（text）为“`Press Me`”，这个文本将出现在按钮上。标记常量（constant）字段，但这时 VDE 已经选择该常量为 `idc_press_me`。该常量将在各种上下文中标识这个特定的控件。

当单击 `OK` 按钮时，该按钮将被插入到这个对话框之中，结果如图 4.12 所示。

现在就可以做一些按钮被单击后要完成的事。为此将打开代码专家（code expert）：右击该对话框，且选择代码专家。代码专家被打开后，可以看到光标正好位于该按钮处，如图 4.13 所示。

在图 4.13 所示的代码专家对话框中，蓝色圆点表示这个控件未被处理。对于处理谓词，VDE 建议使用名字 `onControlPressMe`。右击该项，单击 `add` 按钮，以便将这个谓词添加到代码中，并且将它与这个控件进行绑定。

在单击 `add` 按钮之后，代码专家将发生变化，如图 4.14 所示。

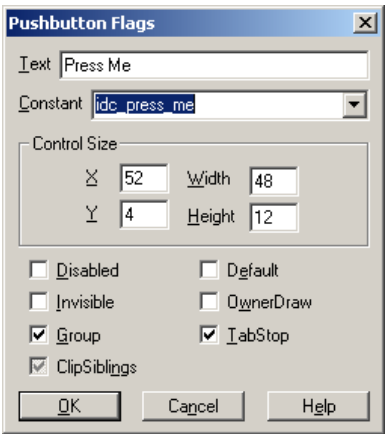


图 4.11 按钮属性标志对话框

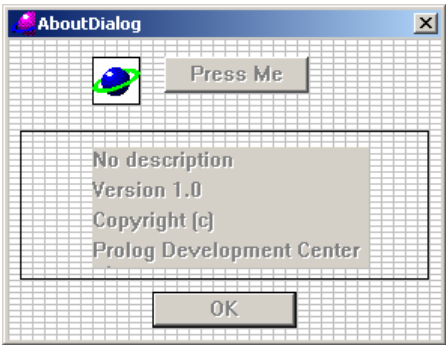


图 4.12 将按钮插入 About 对话框

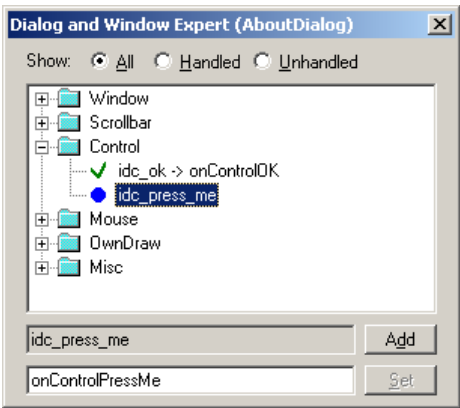


图 4.13 对话框与窗口专家

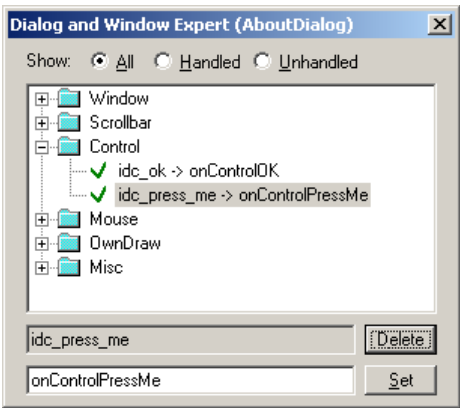


图 4.14 用对话框与窗口专家添加处理谓词

如果在代码专家中双击与该控件对应的行，那么编辑器将被打开，并且定位在新插入的代码处，如图 4.15 所示。

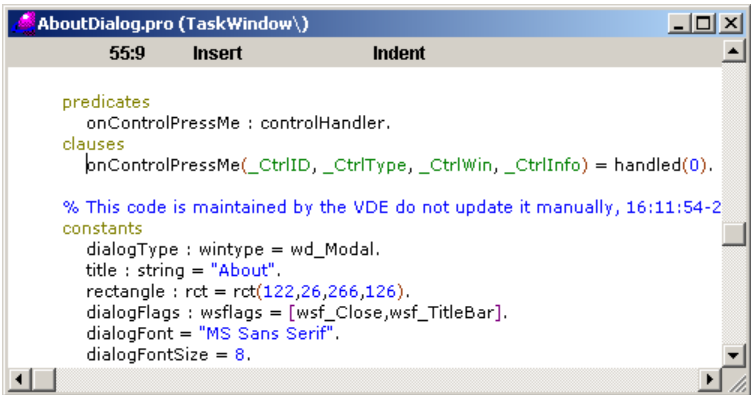


图 4.15 用文本编辑器修改谓词代码

当单击按钮时，可以改变按钮上的文本。为此，修改代码如下：

```
predicates
    onControlPressMe : controlHandler.
clauses
    onControlPressMe(_CtrlID, _CtrlType, CtrlWin, _CtrlInfo) = handled(0) :-
        vpi::winSetText(CtrlWin, "Pressed").
```

注意：变量 `CtrlWin` 在它出现的两个地方被修改为不以下划线开头的变量。否则，编译器将产生一个警告。

现在，试着再次建立和执行该程序（即按下 **F9** 键）。如果已经仔细地遵循上述步骤，到这里应该是成功的。在该程序中，当然应该可以打开 **About** 对话框，然后按下这个新按钮。其工作情形如图 4.16 所示。

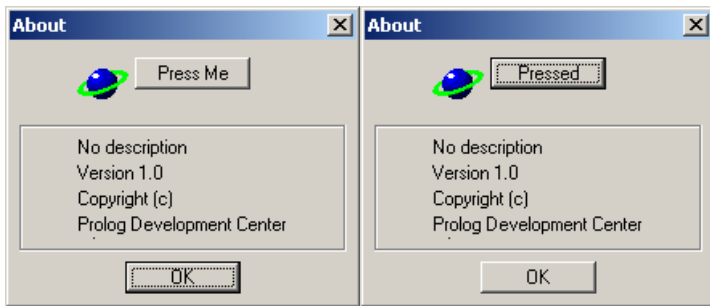


图 4.16 About 对话框按钮工作情形

4.7 调试项目

开发环境包含一个调试器。有了调试器，就可以跟踪程序的执行过程，检查程序的状态。

在学会 **Visual Prolog** 语言之前，详细考查调试器没有太大的意义。但是，在这里仍要简单解释这个调试器，因为它对于理解语言和使用语言是非常有用的。

为了启动调试器，在菜单中选择 **Debug->Run**，或者按下 **F6** 键。如果该项目不是最新的，它将首先被建立，然后才开始调试任务。

注意：通过选择菜单 **Debug -> Stop Debugging** (或者按下 **Shift+F6** 键)，就可以在任何时间停止调试。

调试开始时，**VDE** 将首先装入调试信息，然后进入被调试的程序。程序的执行过程在目标 (goal) 执行之前一直处于停止状态。为了表示这种情形，目标在一个编辑器窗口中被打开，蓝色的箭头指向该目标，如图 4.17 所示。

可以使用调试 (debug) 菜单命令来单步执行程序，如图 4.18 所示。

菜单命令 **step into**：将打开另一个编辑器和代码 `mainExe::run`，箭头将指向这个代码的入口。

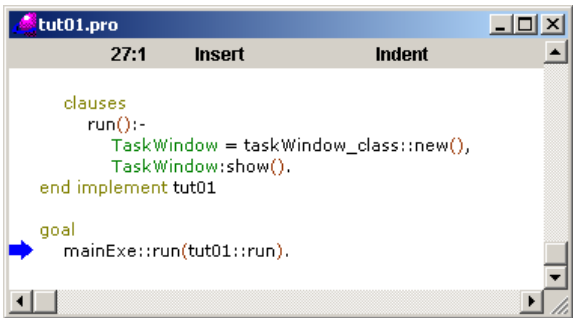


图 4.17 调试程序

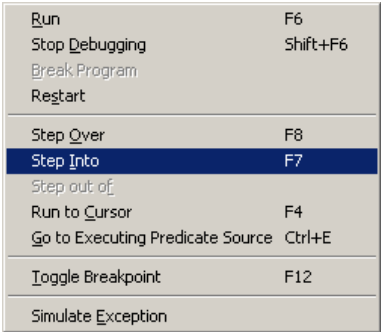


图 4.18 调试器单步执行菜单命令

在视图（view）菜单中，可以打开各种调试器窗口，对此简要做以下解释。

运行堆栈（run stack）窗口：包含一个运行堆栈的描述，如图 4.19 所示。图中所能见到的信息完全取决于在 Tools -> Options -> Debugger 选项中的设置。从原理上讲，运行堆栈包含若干行，与已经进行了的调用相一致。然而，优化（即所谓的尾部调用优化）的结果导致可能已经去掉了一些条目。

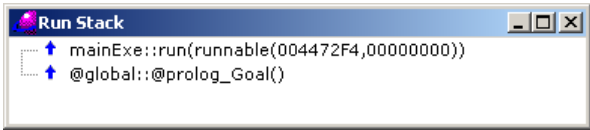


图 4.19 运行堆栈窗口

运行堆栈不仅可以显示调用关系，还可以显示陷入点和回溯点的情况，这些情况将在后续章节里进行更为详细的叙述。

局部变量（local variables）窗口：包含局部变量，与运行堆栈窗口中的选择相对应。一些变量可能尚未接收到值，在这种情形将显示一个下划线，譬如上面的变量 originalraiser，如图 4.20 所示。

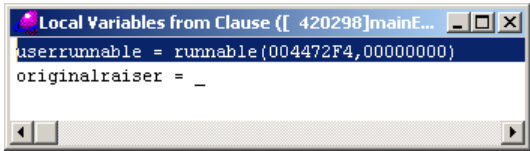


图 4.20 局部变量窗口

事实（facts）窗口：包含程序的全局状态信息，如图 4.21 所示，这个状态存放在事实数据库中。事实数据库是 Visual Prolog 独有的特色，将在后续章节里叙述。也可以从局部变量窗口向事实窗口添加对象。这样，就可以跟踪那些感兴趣的目標的状态变化情况。

断点（break points）窗口：显示程序中当前的断点，如图 4.22 所示。假定已经设置好打开断点，因而该窗口不为空。通过选择菜单 Debug -> Toggle Breakpoint (或按下 F12 键)，可以设置或取消断点。

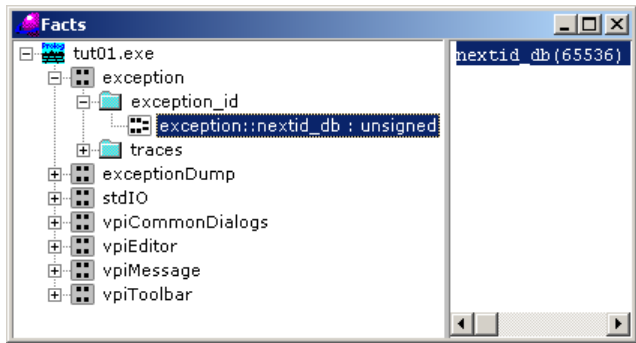


图 4.21 事实窗口

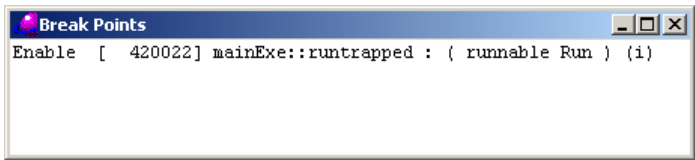


图 4.22 断点窗口

其余的调试窗口属于低级调试，此处将不进行讨论。

本章小结

可视化开发环境（VDE）是人们得以能够创建（create）、建立（build）、维护（maintain）及调试（debug）一个项目的环境。它包含一系列工具、编辑器和向导，来帮助人们完成任务。

习题 4

1. 详细分析 Visual Prolog 与以前的 Prolog 版本有何区别？
2. Visual Prolog 6 有哪些基本特性？
3. Visual Prolog 的可视化开发环境（VDE）有何功用？它包含哪些工具？
4. 调试器有何功用？为什么 VDE 中要包含一个调试器？
5. 通过上机实习，熟悉利用可视化开发环境（VDE）创建、建立以及调试一个项目的方法。

第 5 章 Prolog 基础

本章将介绍关于 Prolog 编程最基本的内容，包括 Horn 子句、Prolog 推理机、程序控制、Prolog 算符等。

Visual Prolog 是面向对象的、严格类型化的和模式检验的程序设计语言。在编写 Visual Prolog 程序时，必须掌握这些内容。但在这里将集中在编写代码这个核心问题上，也就是说，编写这些代码时暂时不考虑类、类型和模式。

为此，将使用包含在 Visual Prolog 6 中的 PIE 例子。PIE 是一个经典的 Prolog 解释器，通过使用它，可以学会和实现 Prolog 程序，而不必关心类、类型等方面的知识。

这里的内容是基于使用 Build 6004 或以后的 Visual Prolog 6 版本，否则，PIE 应用程序将不会像现在描述的这样工作。这个编译号可以在 VDE 的 About 对话框中找到。

5.1 Horn 子句逻辑

Visual Prolog 和其他 Prolog 用语都是基于 Horn 子句逻辑的。Horn 子句逻辑是对事物及其相互关系进行推理的形式系统。

在自然语言中，可以这样表达一个陈述句：

```
John 是 Bill 的父亲.
```

这里涉及两个实体，John 和 Bill，以及他们之间的关系，即一个是另一个的父亲。在 Horn 子句逻辑中，可以这样形式化地表述上面的陈述句：

```
father("Bill", "John").
```

上面的 father 是带有两个参量的一个谓词或关系，它表示第 2 个人是第 1 个人的父亲。

注意：此处已经选择了第 2 个人是第 1 个人的父亲，也可以选择另外的方式，变量的顺序是形式化设计者的选择。然而，一旦选定了，就必须保持一致。在这里的表述中，父亲始终是第 2 个人。

已经选择用人名来代表人。因为在现实世界中，许多人有相同的名字，所以这一方法不一定有效。但在这里，用这一简单的形式化表示。

有了上面的形式化方法，可以表示任何人之间的任何类型的家庭关系。但是，为了让这些表述更为有趣，制定下面的规则：

```
x 是 z 的祖父，如果 x 是 y 的父亲 且 y 是 z 的父亲
```

其中 X, Y, Z 指人。在 Horn 子句逻辑中，可以这样表述：


```
grandFather(Person, GrandFather):-
    father(Person, Father), father(Father, GrandFather).
```

已经选择使用了比 **X**, **Y**, **Z** 更容易理解的变量名。另外, 还引入了一个谓词来描述祖父关系。再次选择祖父作为第 2 个变量, 像这样保持一致是明智的, 不同谓词的变量可以遵循相同的规则。当解读这些规则时, 可以将 “:-” 解释为 “如果 (if)”, 将隔开关系的逗号解释为 “与 (and)”。

像 “John 是 Bill 的父亲” 这样的陈述称为事实, 而 “X 是 Z 的祖父, 如果 X 是 Y 的父亲且 Y 是 Z 的父亲” 称为规则。

可以用事实和规则来形成定理, 一个定理是事实和规则的集合。下面陈述一个小定理:

```
father("Bill", "John").
father("Pam", "Bill").

grandFather(Person, GrandFather):-
    father(Person, Father),
    father(Father, GrandFather).
```

这个定理的作用是回答这样一些问题:

```
John 是 Sue 的父亲吗?
谁是 Pam 的父亲?
John 是 Pam 的祖父吗?
...
```

这些问题称为目标 (goal)。它们可以这样形式化表述:

```
?- father("Sue", "John").
?- father("Pam", X).
?- grandFather("Pam", "John").
```

这些问题被称为目标子句 (goal clause) 或简称为目标。事实 (facts)、规则 (rules) 及目标合起来称为 Horn 子句, 因此得名 Horn 子句逻辑。

某些目标, 如第 1 个和最后一个目标, 可以简单地用 “是” 或 “不是” 来回答。其他目标, 如第 2 个目标, 需要寻找一个解, 例如, **X** = “Bill”。

一些目标可能有多个解, 例如,

```
?- father(X, Y).
```

有两个解:

```
X = "Bill", Y = "John".
X = "Pam", Y = "Bill".
```

一个 Prolog 程序是一个定理和目标的集合。当程序开始时, 它试图使用定理为目标找到一个解。

5.2 Prolog 推理机

现在试一下 PIE 中的小例子。PIE (Prolog inference engine)即 Prolog 推理机，随 Visual Prolog 6 一起提供。

在开始之前，必须先安装和建立 PIE 的例子：

(1) 在 Windows 开始菜单中选择“安装例子”(Start -> Visual Prolog 6 -> Install Examples)。

(2) 用 VDE 打开 PIE 项目，并运行该程序。

当程序启动后，其情形如图 5.1 所示。

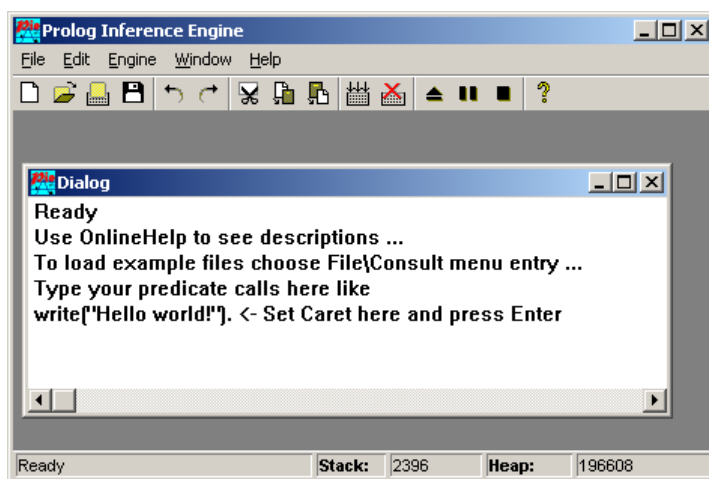


图 5.1 Prolog 推理机

选择 File -> New，键入前面的父亲子句和祖父子句，如图 5.2 所示。

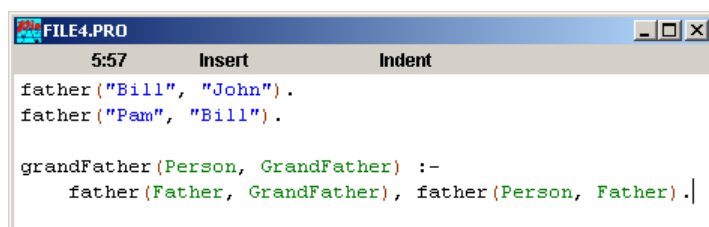


图 5.2 子句代码

当编辑器窗口激活时，选择 Engine -> Reconsult，这将会把文件装入到推理机。在对话框中，还将得到这样一个消息：

```
Reconsulted from: ....\pie\Exe\FILE4.PRO
```

无论用编辑器如何装入，其内容都不会保存到文件之中。如果想要保存内容，必须使

用菜单命令 **File -> Save**。

菜单 **File -> Consult** 不管文件是否因编辑而打开，都会装载磁盘文件中的内容。

一旦查阅过定理，就可以回答各种目标。

在对话框窗口的空白行上，键入一个目标，不带前缀“?-”。例如，键入如图 5.3 所示的查询代码。

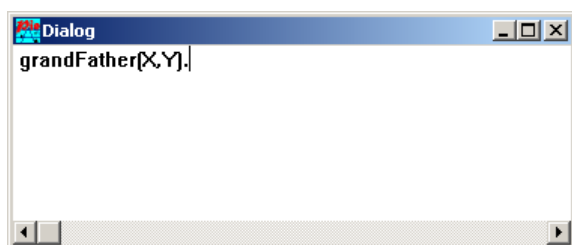


图 5.3 键入目标

当插字符号位于行尾时，按下键盘上的 **Enter** 键。PIE 现在将把从该行的开头到插字符号之间的文本当作目标来执行。于是，可以看到如图 5.4 所示的结果。

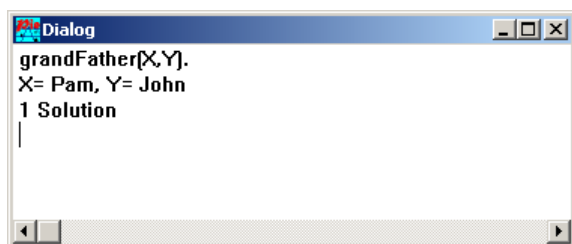


图 5.4 查询目标对话框

5.3 扩展家庭定理

使用诸如 **mother** 和 **grandMother** 这样的谓词，可以直接扩展家庭定理。读者应该试着亲自去做，也可以试着添加更多的人。建议读者使用自己家庭中的人，因为这样易于验证，且可以不考虑添加的这个人是否真正是自己的祖母。

给出谓词 **mother** 和 **father**，还可以定义双亲 (**parent**) 这个谓词。一位母亲是双亲，一位父亲也是双亲，因此可以使用两个子句来定义双亲：

```
parent(Person, Parent) :- mother(Person, Parent).  
parent(Person, Parent) :- father(Person, Parent).
```

第 1 个规则可以解读为：如果 **Parent** 是 **Person** 的 **mother**，则 **Parent** 是 **Person** 的双亲 (**parent**)。

还可以用分号“;”来定义双亲 (**parent**) 关系，分号代表“或 (or)”。

```
parent(Person, Parent) :-  
    mother(Person, Parent);  
    father(Person, Parent).
```

这条规则可以解读为: 如果 Parent 是 Person 的 mother 或(or)Parent 是 Person 的 father, 则 Parent 是 Person 的双亲 (parent)。

强烈建议读者尽可能少用或根本不用这个分号“;”。之所以这样建议, 是基于以下理由:

(1) 逗号“,”和分号“;”之间在印刷上的差别非常小, 但语义上的差别却很大。分号“;”常常是引起混淆的一个根源, 因为它容易被误解为逗号“,”, 特别是当它处于一个长行的末尾时。

(2) Visual Prolog 只允许在最外一层使用分号 (PIE 允许任意层次的嵌套)。

试着生成一个同胞 (sibling) 谓词。可以发现同胞被找到两次。如果两个人有同一个母亲, 则他们是同胞; 如果两个人有同一个父亲, 则他们也是同胞。也就是说, 可以建立如下规则:

```
sibling(Person, Sibling) :- mother(Person, Mother), mother(Sibling, Mother).  
sibling(Person, Sibling) :- father(Person, Father), father(Sibling, Father).
```

第 1 个规则可以解读为: Sibling 是 Person 的同胞, 如果 Mother 是 Person 的母亲, 且 Mother 是 Sibling 的母亲。

之所以会找到同胞关系两次, 是因为大多数同胞都有相同的父亲和母亲, 满足上述必要条件, 因此才会被找到两次。

现在不解决这一问题, 当前只认可这一点, 一些规则将产生多个结果。

一个完全血缘关系 (fullBlodedSibling) 谓词没有同样的问题, 因为它需要有相同的父亲和母亲。

```
fullBlodedSibling(Person, Sibling) :-  
    mother(Person, Mother),  
    mother(Sibling, Mother),  
    father(Person, Father),  
    father(Sibling, Father).
```

5.4 Prolog 是一种编程语言

从以上描述可以发现, 到目前为止, 读者会认为 Prolog 是一个专家系统, 而不是一种程序设计语言。的确, Prolog 可以作为专家系统来使用, 但它本身却是作为一种程序设计语言而设计出来的。

把 Horn 子句逻辑变为一种程序设计语言的两个重要因素:

- (1) 严格的搜索顺序或程序控制;
- (2) 副效应。

5.5 程序控制

当试图为如下目标寻找一个解时：

```
?- father(X, Y).
```

可以有許多实现方式。例如，如果只考虑定理中的第2个事实，那么就可得到一个解。

但是 Prolog 不使用随机搜索策略，而总是使用同一种策略。系统保持一个当前目标，始终从左到右进行求解。

例如，如果当前目标是

```
?- grandfather(X, Y), mother(Y, Z)
```

那么系统就会尝试在求解子目标 `mother(Y, Z)` 之前，首先求解子目标 `grandfather(X, Y)`。如果第1个（即最左面的）子目标不能被求解，则全部问题就没有解，第2个子目标根本就不用再尝试求解。

当求解一个特定子目标时，事实和规则将被自上而下进行尝试。

当使用规则求解了一个子目标时，当前目标中待求解的那个子目标将被其规则右边的那些子目标所代替。

例如，如果当前目标是

```
?- grandfather(X, Y), mother(Y, Z).
```

而使用规则

```
grandfather(Person, GrandFather):-  
    father(Person, Father), father(Father, GrandFather).
```

去求解第1个子目标，则作为结果的当前目标是

```
?- father(X, Father), father(Father, Y), mother(Y, Z).
```

注意，规则中的某些变量已经被子目标中的变量替换，在后面将详细解释这一点。

给定这种评价策略，就可以更多地从程序上解释子句的控制过程。

考虑这个规则：

```
grandfather(Person, GrandFather) :-  
    father(Person, Father), father(Father, GrandFather).
```

给定严格的评价策略，可以这样解读这个规则：为了解 `grandfather(Person, GrandFather)`，首先要求解 `father(Person, Father)`，然后再求解 `father(Father, GrandFather)`。或者这样理解：当调用 `grandfather(Person, GrandFather)` 时，首先要调用 `father(Person, Father)`，然后再调用 `father(Father, GrandFather)`。

有了这种程序上的解读，就可以理解，所谓的谓词实际上就相当于其他编程语言中的

过程或子例程。它们之间主要的区别在于一个 Prolog 谓词对于一个单个提问可以返回多个结果或没有结果（即失败）。这一点将在 5.6 节详细进行讨论。

5.5.1 失败

在定理中，一个谓词提问可能没有任何一个解。例如，调用 `parent("Hans", X)`，因为不存在适用于 Hans 的双亲关系的事实或规则，所以没有解。通常说这个谓词调用失败（fail）。如果目标失败了，则说明定理中完全不存在针对该目标的解。5.6 节将解释在通常情况下，即当它不是一个失败的目标时，如何处理失败。

5.5.2 回溯

在 Prolog 程序的过程性解释中，“或（or）”以相当特殊的方式进行处理。考虑下面的子句：

```
parent(Person, Parent):-  
    mother(Person, Parent);  
    father(Person, Parent).
```

从逻辑上解读，这样解释这个子句：如果 Parent 是 Person 的母亲，或（or）Parent 是 Person 的父亲，则 Parent 是 Person 的双亲。

对于 Parent 谓词的提问，“或（or）”引出了两个可能的解。Prolog 处理这种多项选择时，往往首先尝试第 1 个选择，随后根据需要，再回溯到下一个备份的选择等。

在程序执行期间，来自早期谓词调用的许多备份的选择（称为回溯点）可能继续存在。如果有的谓词调用失败了，那么将回到上一个回溯点尝试另一个选择性的解。如果没有回溯点存在了，则整个目标就失败了，也意味着没有解存在。

这样就可以这样解释上面的子句：当 `parent(Person, Parent)` 被调用时，首先给第 2 个选择性的解（即对于调用 `father(Person, Parent)`）记录一个回溯点，接着再调用 `mother(Person, Parent)`。

一个谓词可以有几个类，其行为方式类似。考虑下面的子句：

```
father("Bill", "John").  
father("Pam", "Bill").
```

当 `father` 被调用时，首先给第 2 个子句记录一个回溯点，然后尝试第 1 个子句。

如果有 3 个或更多的选择，仍然只创建一个回溯点，但这个回溯点将通过创建另一个回溯点而开始。考虑下面的子句：

```
father("Bill", "John").  
father("Pam", "Bill").  
father("Jack", "Bill").
```

当 `father` 被调用时，首先记录一个回溯点，然后尝试第 1 个子句。所创建的这个回溯

点指向一些代码，这些代码本身创建一个回溯点（即给第3个子句），然后尝试第2个子句。因此，所有的选择点只有两个选择，但一个选择本身可能包含着另一个选择。

为了阐明程序如何执行，详细考查一个例子。考虑下面这些子句：

```
mother("Bill", "Lisa").

father("Bill", "John").
father("Pam", "Bill").
father("Jack", "Bill").

parent(Person, Parent):-
    mother(Person, Parent);
    father(Person, Parent).
```

然后考虑目标：

```
?- father(AA, BB), parent(BB, CC).
```

这个目标表明，要找3个人AA，BB和CC，其中BB是AA的父亲，CC是BB的双亲。

如上所述，总是从左到右求解目标，所以会先调用father谓词。当执行father谓词时，首先为第2个子句创建一个回溯点，然后执行第1个子句。

通过第1个子句发现，AA是Bill，BB是John，所以现在实际上又有了下面的目标：

```
?- parent("John", CC).
```

于是调用parent谓词，它又给出下面的目标：

```
?- mother("John", CC); father("John", CC).
```

注意，子句中的变量已经被调用中的实际参数所替换（与其他编程语言中调用子例程时的情形完全一样）。

当前目标是一个“或(or)”目标，所以为第2个谓词创建一个回溯点，然后执行第1个谓词。现在有两个活动的回溯点，一个是parent子句中的第2个谓词选项，另一个是father谓词中的第2个子句。

在创建这个回溯点之后，遇到如下目标：

```
?- mother("John", CC).
```

于是调用mother谓词。当第1个参数是John时，mother谓词失败（因为没有子句在第1个参数匹配这个值）。

当谓词失败时，回溯到所创建的上一个回溯点。所以现在将寻求目标：

```
?- father("John", CC).
```

当这次调用father谓词时，将再次首先为第2个father子句创建一个回溯点。还有一个回溯点，它是为father谓词的第2个子句所创建的，它对应于原始目标中的第1次调用。

现在使用目标中的第 1 个 `father` 子句, 由于第 1 个参数不匹配, 即 `John` 与 `Bill` 不匹配, 所以失败了。

因此回溯到第 2 个子句, 但是在使用这个子句前, 已为第 3 个子句设置了一个回溯点。

第 2 个子句也失败了, 因为 `John` 与 `Pam` 不匹配。所以回溯到第 3 个子句, 同样也失败了, 因为 `John` 与 `Jack` 不匹配。

现在必须完全回溯到原始目标中第 1 个 `father` 调用, 在这里为第 2 个 `father` 子句创建了一个回溯点。

使用第 2 个子句, 发现 `AA` 是 `Pam`, `BB` 是 `Bill`, 所以现在的实际目标是

```
?- parent("Bill", CC).
```

当调用 `parent` 时, 得到

```
?- mother("Bill", CC); father("Bill", CC).
```

再次为第 2 个选项创建一个回溯点, 然后执行第 1 个谓词子句:

```
?- mother("Bill", CC).
```

当 `CC` 是 `Lisa` 时, 这个目标成功了, 于是为该目标找到了一个解:

```
AA = "Pam", BB = "Bill", CC = "Lisa".
```

当试图寻找另外的解时, 回溯到上一个回溯点, 就是 `parent` 谓词的第 2 个选项:

```
?- father("Bill", CC).
```

当 `CC` 是 `John` 时, 目标成功, 所以又发现目标的另一个解:

```
AA = "Pam", BB = "Bill", CC = "John".
```

如果接着尝试, 将会得到更多的解:

```
AA = "Jack", BB = "Bill", CC = "John".
```

```
AA = "Jack", BB = "Bill", CC = "Lisa".
```

最后所有的尝试都将失败, 没有留下任何回溯点, 所以这个目标总共有 4 个解。

5.5.3 改进家庭定理

如果继续处理上述家庭关系, 就可能会发现难于处理诸如兄弟、姐妹这样的关系, 这是因为要确定一个人的性别是相当困难的, 除非这个人是一位父亲或母亲。

问题是选择了一个不好的方式来形式化这个定理。原因是从考虑实体间的关系开始。如果首先考虑实体本身, 那结果就会不同。

主要实体是人, 人都有名字 (在这个简单的例子中, 仍然用名字来标识人。而在一个真正有规模的程序中, 这一点未必成立)。人有性别。人还有许多其他的特性, 但与我们的兴趣无关, 所以没有出现在上下文中。

由此这样定义一个 `person` 谓词：

```
person("Bill", "male").
person("John", "male").
person("Pam", "female").
```

`person` 谓词的第 1 个参数是名字，第 2 个参数是性别。

不是使用 `mother` 和 `father` 作为事实，而是选择 `parent` 作为事实，将 `mother` 和 `father` 作为规则：

```
parent("Bill", "John").
parent("Pam", "Bill").

father(Person, Father) :- parent(Person, Father), person(Father, "male").
```

注意：当 `father` 是一个“导出”关系时，不可能有女性的父亲。所以这个定理在内部要保持一致，在这一点上不能有其他的形式。

5.5.4 递归

利用上面给出的原则，大多数家庭关系是容易建立的。但当它涉及像祖先这样的“无穷”关系时，则将需要更多的规则。如果遵循上述原则，就应该这样定义祖先：

```
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), parent(P1, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), parent(P1, P2), parent(P2, Ancestor).
...
```

主要问题是子句的排列永远不会完结。克服这一问题的方法是使用一个递归定义，即像下面这样，根据它自身进行的定义：

```
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), ancestor(P1, Ancestor).
```

这一声明指双亲是一个祖先，双亲的祖先还是祖先。

如果还没有熟悉这种递归的概念，就可能会觉得递归很难处理。递归是 `Prolog` 程序的基础，需要反复加以使用，然后才会习惯。

试着执行一个祖先 `ancestor` 目标：

```
?- ancestor("Pam", AA).
```

为第 2 个 `ancestor` 子句设置一个回溯点，然后执行第 1 个子句。这时，新的目标为

```
?- parent("Pam", AA).
```

这样很快找到一个解：

```
AA="Bill".
```

然后，试图通过使用第 2 个 `ancestor` 子句的回溯点来找寻另一个解。这时，新的目标为

```
?- parent("Pam", P1), ancestor(P1, AA).
```

由于 Bill 是 Pam 的双亲，所以找到 `P1="Bill"`。接着，目标为

```
?- ancestor("Bill", AA).
```

为了求解这个目标，首先为第 2 个 `ancestor` 子句设置一个回溯点，然后执行第 1 个子句。这时给出下列目标：

```
?- parent("Bill", AA).
```

这个目标又给出一个解：

```
AA = "John".
```

所以找到 Pam 的两个祖先：Bill 和 John。

如果使用第 2 个 `ancestor` 子句的回溯点，将得到下列目标：

```
?- parent("Bill", P1), ancestor(P1, AA).
```

在这里，再次发现 John 是 Bill 的双亲，因此 `P1` 为 John。这又给出了目标：

```
?- ancestor("John", AA).
```

如果继续求解这个目标，将发现这个目标没有任何一个解存在。所以，归根到底只能找到 Pam 的两个祖先。

递归具有非常强大的功能，但它也有一点难于控制。使用递归时，切记以下要点：

- 递归必须能够前进；
- 递归必须能够终止。

在上面的代码中，第 1 个子句确保该递归能够终止，这是因为这个子句不是递归的，即它没有对该谓词本身进行调用。第 2 个子句是递归的。在第 2 个子句中，保证在进行递归调用之前，更深一层地回退一个祖先步。也就是说，确保在问题求解过程中将不断取得一些进展。

5.5.5 副效应

除了严谨的计算顺序以外，Prolog 也有一些副效应。例如，Prolog 有一些用于读写操作的预定义谓词。

下面的目标将输出找到的 Pam 的祖先：

```
?- ancestor("Pam", AA), write("Ancestor of Pam : ", AA), nl().
```

`ancestor` 调用将找到 Pam 的一个祖先，并放在 `AA` 中。

`write` 调用将输出字符串 "Ancestor of Pam : "，然后输出 `AA` 的值。

`nl` 调用将会使输出转入到一个新行。

当在 **PIE** 中运行程序时，**PIE** 自己会给出结果。所以会造成自己的结果和 **PIE** 的结果混在一起，所得出的结果可能并不是想要的。

避免 **PIE** 本身输出结果的一个非常简单的方法就是确保该目标没有解。考虑下面的目标：

```
?- ancestor("Pam", AA), write("Ancestor of Pam : ", AA), nl(), fail.
```

`fail` 是一个预先定义的谓词，它总是失败，即不会有解。

前 3 个谓词调用和上面的效果完全相同：若一个祖先被找到（如果存在的话），则把它写出来。接着调用 `fail`，程序失败。因此如果可能，我们应追寻一个回溯点。

在追寻该回溯点时，找到另一个祖先（如果存在的话），把它写出来，然后再失败，如此反复。

于是找到并且输出所有的祖先，直到最后不再有回溯点，目标执行失败。

这里有以下要点需要加以注意：

- 目标本身不存在一个单一的解，从而使想要的全部解都作为副效应形式给出；
- 副效应在失败计算中也存在。

这些情形是一个事物的两个方面，但它们代表了不同阶层的观点。第 1 个阶层乐观地表明可以利用的一些可能性；第 2 个阶层比较悲观，提醒要注意利用副效应，因为即使当前目标不产生任何解，它们照样也会完成。

任何人学习 **Prolog** 时，迟早都会经来自部分失败程序的意外输出的情况。也许下面这个建议可以对此有所帮助：将计算性代码与执行输入输出的代码分开。

在上面的例子中，所有的谓词都是计算谓词。它们会计算出一些家庭关系。如果需要把解（例如，“双亲”）写出来的话，就构造另外一个谓词来写出双亲，而让这个谓词调用计算双亲的谓词。

5.5.6 小结

这一节介绍了 **Prolog** 的一些基本特性、事实、规则和目标以及 **Prolog** 的执行策略，包括失败和回溯。同样发现回溯可以给出多个结果，最后介绍了副效应。

5.6 Prolog 算符

本节继续介绍 **Prolog** 编程的基本思想。本节主要关心 **Prolog** 中的数据在被操作之前是怎样被模型化的。因此，并没有太多关于代码执行的例子。这里假定读者已经熟悉了执行策略（**execution strategy**）对结果的影响和副效应（**side effects**）是怎样把一个 **Prolog** 程序的逻辑转化为所需结果的。

与 5.5 节一样，将继续使用 **PIE** 环境来学习 **Prolog**，只有在本书的后部分内容才有必要进入 **Visual Prolog 6** 可视化开发环境（**VDE**）。

5.6.1 算符

5.5 节中，所有的人都用名字 Bill、John 和 Pam 等表示。现在，Bill、John 和 Pam 惟一代表各个人的名字。这些名字的值是简单数据类型或简单论域（simple domains）。作为人名时，这种简单论域的类型是字符串，其他的简单论域可以是数（如 123 或 3.14）、符号（如 xyz 或 chil_10）和字符（如 5 或 c）。

然而，人是由包含名字在内的更多特性所表示的，当需要表达所有特性而不仅仅是名字时怎么办呢？也就是说，需要某些机制来表示复合论域（compound domains）。它是更为简单论域的一个集合。

5.5 节中，曾试着通过向 PIE 系统加入集中于实体（entities），而不是关系的事实把个人的多种特征放在一块儿，比如名字 name 和性别 gender，从而给出了如下事实：

```
person("Bill", "male").
person("John", "male").
person("Pam", "female").
```

然而，有一个更优雅的方法可以让人们把注意力集中在被表示的实体上。

可以将姓名和性别用被称为复合论域的形式封装在一个包中，然后整个封装可以用一个 Prolog 子句中的一个逻辑变量来表示，像其他任意变量一样。例如，上述事实可以被一般地表示如下：

```
person(Name, Gender)
```

注意，上面的句子既不是一个事实也不是一个谓词。逻辑上，它表示程序中的名为 person 的复合论域，每个 person 有两个特征，由逻辑变元 Name 和 Gender 表示。单词 person 就是所谓的算符，变元 Name 和 Gender 就是它的参数。以后应用这些算符来封装事实。

由于复合论域总有算符，以后将用算符来代表各种复合论域。

现在，用新定义的算符 person 改变上节的第 1 个例子，如图 5.5 所示。

注意，在使用的 PIE（Prolog 推理机）中，可以直接使用复合论域而不必事先声明（比如说，不必为了让代码运行而定义一个名为 person 的谓词或事实）。

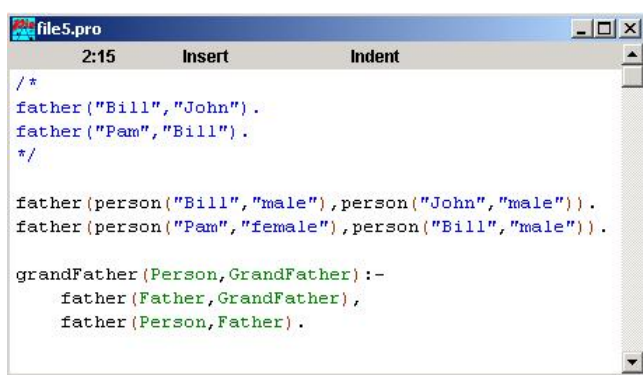


图 5.5 person 算符

观察表示 `father` 关系的两个事实，可以发现人的定义比以前更丰富了（原来的代码已经被注释掉了——在 `/*` 和 `*/` 之间）。这时，每个人使用 `person` 算符描述他们的名字和性别，而在 5.5 节，只使用了人的名字。

更改代码后，使用 `Engine -> Reset`，确保 `PIE` 已经复位。然后，用 `Engine -> Reconsult` 重新求解。

以前，在对话窗口中的空白行上输入一个目标（前面不带“？”号），如图 5.6 所示。

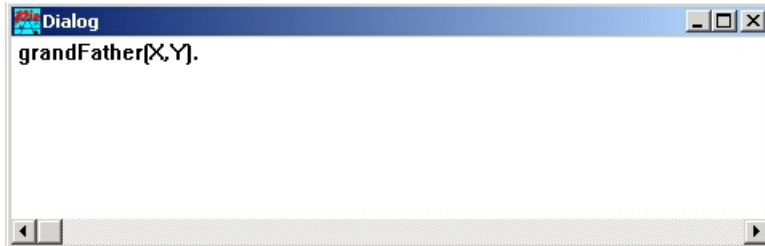


图 5.6 目标代码

在行尾输入完句号后，按 `Enter` 键，`PIE` 将把行首到句号的文本当作目标来执行，然后就可以看到如图 5.7 所示的结果。

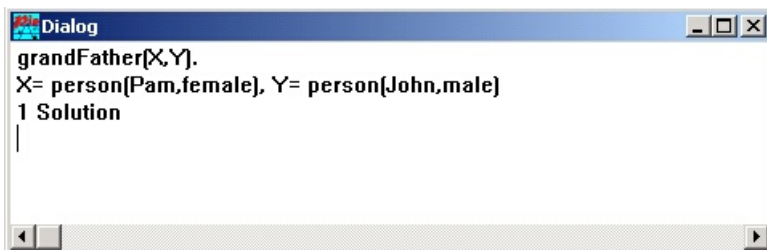


图 5.7 目标代码结果

可以发现，现在得到的结果比以前更丰富了。

5.6.2 深入理解算符

看谓词 `grandfather`，就会发现一个微妙的问题，一个人应该有两个祖父：一个来自爸爸那边，一个来自妈妈那边，但按前面定义的谓词 `grandfather`，只能得到从爸爸那边来的祖父。

因此，谓词 `grandfather` 应该重写为

```
grandFather(Person, TheGrandFather) :-
    parent(Person, ParentOfPerson),
    father(ParentOfPerson, TheGrandFather).
```

这个谓词逻辑考虑到了任一双亲的父亲都是祖父的常识。

为了让它工作，需要再定义一个用到算符 `person` 的谓词 `father`。这个谓词将遍历系统

中定义的 `parents` 事实数据库。对于找到父亲们来讲，这是一个比把他们作为事实列出来（如前所示）更为明智的做法。因为通过后者可以用类似的形式来扩展概念，找出母亲。

可以写成下面两种形式中的任意一种：

```
/*1st version */

father(P, F) :-
    parent(P, F),
    F = person(_, "male").    %Line 2
```

或者

```
/* 2nd version */

father(P, person(Name, "male")):-
    parent(P, person(Name, "male")).
```

以上两个版本的逻辑是相同的，但交给 Prolog 推理机的方法不同。第 1 个版本中，Prolog 依次检查代码中的 `parent` 事实，并看其是否与从谓词头传递来的第 1 个逻辑变量(P)匹配。如果该变量确实匹配，则检查第 2 个参数是不是 `person` 算符，且 `person` 算符的第 2 个参数是不是字符串 `male`。

这个例子体现了算符的一个重要特性：一个算符的多个参数可以通过常见的 Prolog 变量和绑定值被分离和检查（就像本例中的字符串精确匹配）。在第 2 行（第 1 个版本中用“%Line 2”注释的地方），会发现仍用了一个匿名变量（下划线）作为 `person` 算符的第 1 个参数，因为不关心 `father` 具体的名字。

第 2 个版本与第 1 个版本功能相同。但在这里，检查双亲事实集合的同时，找到正确的 P 值，就停下来并返回正确的算符给谓词头，若该谓词的第 2 个参数是字符串 `male`，则该算符不再绑定到任何 Prolog 中间变量，但在第 1 个版本不是这样。

第 2 个版本比第 1 个要简洁，但有时这种写法对初学者来说可能更难读一些。

现在，可以在一个具有 `person` 事实的完整例子中使用这些代码，如图 5.8 所示。

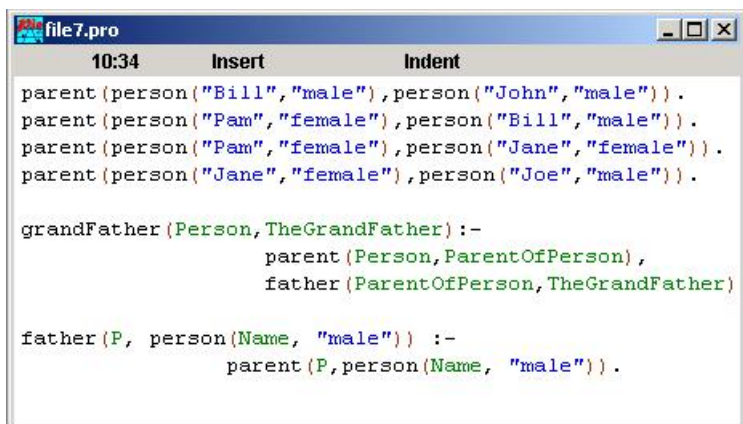


图 5.8 目标代码结果

把上面的代码输入 PIE 中，并给出如下目标：

```
grandFather(person("Pam","female"),W)
```

结果如图 5.9 所示。

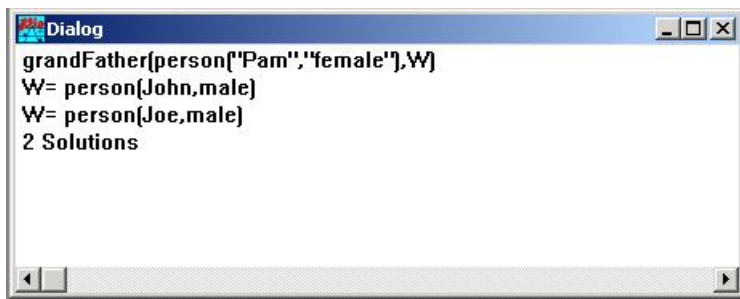


图 5.9 目标代码结果

5.6.3 算符与谓词

从技术角度讲，一个算符代表一个将多个论域绑定到一块儿的逻辑功能。也就是说，算符是一种使 Prolog 推理机把数据的各个部分放在一起的一种机制，它有效地把数据的各部分放在一个通用的盒子中。在需要时，像前面的例子，也可以获得其成分。它看起来也许像一个 Prolog 事实或一个谓词调用，但它不是，它只是一份数据，一个在很大程度上可以像一个字符串或数一样操作的数据。

注意，算符与其他编程语言中的函数毫无关系。它不能够进行运算。它只不过简单地代表复合论域并把自身的参数集成在一起。

通过研究上面的例子，发现用逻辑变量代表由算符表示的数据时无需做特别的操作。表示这些数据的变量可以像其他任何变量一样书写：以大写字母开头。因此，体会本节例子中的 `grandFather` 谓词，就会发现它与前面一节中定义的同一谓词完全一样，毕竟该谓词的逻辑没有改变。所以，该谓词中所用的变量也就没有任何变化。

使用算符的最大好处在于，修改代码时可以自由地改变算符的内部参数，而对使用该算符的谓词无需做大的改动。

也就是说，如果想让 `person` (在后续版本中) 有更多的参数，仍然无需对谓词 `grandFather` 做任何改动。

5.6.4 算符作为参数

5.5 节中，算符 `person` 有两个参数：Name 和 Gender。两者正好都是像常量 `Bill` 和 `Male` 这样的简单论域。然而，也可以把一个算符作为另一个算符的参数。

假定想为一对夫妇（丈夫和妻子）定义一个算符，就可以使用这样一个算符：

```
myPredicate(ACouple):-
```

```
ACouple=couple(person(Husband,"male"),
               person(Wife,"female")
               ), ...
```

本例中，发现这个算符用另外两个算符来定义，每个算符都是变量和常量的混合体。这正好反映了数据被描述的逻辑。这里所用的逻辑是丈夫总是 **Male**，而妻子总是 **Female**，一对夫妇由一个丈夫和一个妻子组成。所有这些都与通常的夫妇概念相一致。

尽管在 **PIE** 中，无法预定义算符的实际语法类型，但在 **Visual Prolog** 中可以这样定义。这样定义一个算符的好处在于，当用实际数据将算符实例化时，**Prolog** 推理机将总是检查其类型的一致性。

这引出了算符的另一重要特性：算符 **couple** 由两个参数定义且参数的位置反映了他们的逻辑结合关系。在 5.1 节中已经解释过了，谓词参数的位置由设计代码的程序员给出。一经给出，就必须总是采用这样的位置形式。

该法则同样适用于算符的构造。例如，算符 **couple**，恰好把 **husband** 作为第 1 个参数，**wife** 作为第 2 个参数。一旦这样做，以后无论何时用到此算符，就必须保证 **husband** 总在前面，而 **wife** 总在后面。

现在回忆算符 **couple**，有人会说，如果不能保证第 1 个参数总是一个丈夫（总是男性），且第 2 个总是一个妻子（女性），那应该怎样定义它的参数呢？这时，可定义一个更简化的 **couple** 算符如下：

```
myPredicate(ACouple):-
    ACouple=couple(Husband,Wife), ...
```

因此，可以把 **husband** 和 **wife** 颠倒过来用，但这次是在不知道哪个是 **husband** 的位置，哪个是 **wife** 的位置的情况下使用的。

```
myPredicate(Couple):-
    Couple=couple(person(PersonsName,PersonsGender),
                  person(SpouseName,SpouseGender)
                  ),...
```

对于上面的算符，下面的两个例子都具有逻辑意义：

```
myPredicate(C1):-
    C1=couple(person("Bill", "male"),person("Pam", "female")),...

/* or */
myPredicate(C2):-
    C2=couple(person("Pam", "female"),person("Bill", "male")),...
```

需要指出的是，在 **PIE**（和其他许多 **Prolog** 推理机）中，通过算符的定义，无法看出该算符是接受简单论域还是复合论域。这是由于在 **PIE** 中，复合论域可以不经声明而直接使用，这与程序员的想法不同。

例如，要用下面的复合论域：


```
person(Name,Gender)
```

那么 PIE 将很容易接受像下面这样的逻辑变量：

```
regardingAPerson(Somebody):-
    Somebody=person("Pam",
                    person("Pam",
                          person("Pam",
                                person("Pam","female")
                              )
                        )
    ), ...
```

实际上，这在当前上下文中没有任何逻辑意义。幸运的是，Visual Prolog 在区分简单论域和复合论域方面做了许多很好的工作，所以看了后面的教程，读者就不会有这些问题了。

5.6.5 算符递归

使用算符描述数据时，可以像其他任何数据一样操作。例如，前面用到的谓词 `ancestor`。

为了确定某人是否是另一个人的先辈，使用递归定义了该谓词，也就是说，引用了自身的定义，如下：

```
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), ancestor(P1, Ancestor).
```

这个定义表示，父母是先辈，并且父母的先辈也是先辈。可以发现，上面的变量可以很好地代表简单论域或复合论域。

因而这样定义数据：

```
parent(person("Bill","male"),person("John","male")).
parent(person("pam","female"),person("Bill","male")).
```

如果现在让 PIE 求解如下目标：

```
P=person("pam","female"),ancestor(P,Who)
```

将得到如图 5.10 所示的结果。

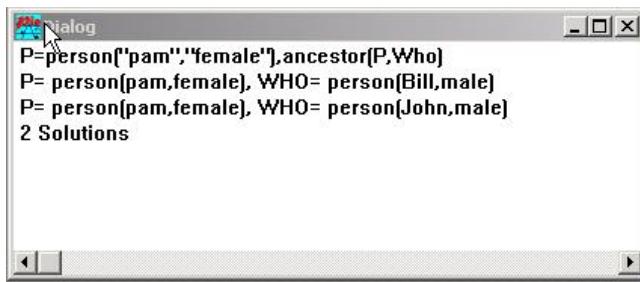


图 5.10 目标代码结果

PIE 推理机循环检查 `parent` 事实来找到可能的答案。在上面的求解中，会发现在给 PIE 指定目标时将复合论域 `person` 绑定到变量 `P`，其目的是用来让代码更易读，但同时也演示了可以将一份数据指定为复合论域赋给任何 Prolog 变量。

5.6.6 算符使用策略

有了模型化的数据，软件才会尽可能的好。建模就是建立外部真实世界和软件内部数据结构的关系。如果数据模型很差，软件一般来说也很差；或者最多也是没有效率。这对于任何的编程语言和任何软件都是成立的。这也正是在前面努力把注意力集中在实体上，并尽量不使用更多事实的原因。类似地，本节介绍了算符的概念，以使被模型化的实体的数据更清晰。

Prolog 的好处在于，它可以用一种内部代码能够高效使用的形式来描述客观数据。同时，它使代码对于同一项目组中的其他程序员更易读。

建模过程中，算符可以用来帮助建立任何类型的复合论域。在软件开发的关键阶段，必须仔细观察真实世界的多样性来设计步骤并用算符（和后面的内容中要碰到的其他数据类型）转化它们，牢记它们的用法。

当把注意力集中到软件中要用到的算符（或其他数据结构）上时，首先对它应该有一个全面的认识。然后暂停下来审视软件中的每个细节。这样就能写出确实需要的算符。

仔细构造数据结构并不是惟一的要求，还需要（常常是同时的）写或更改不同的目标及子目标（也就是谓词），从而充分应用那些谓词中的数据。为了达到这些目标和子目标的副效应会让软件正确运行的目的，还可以从中得到有价值的反馈来改进自己的数据结构。

本节没有设计一个完整的软件。只是用一点点数据来建立对真实世界中人与人（父母、祖父、家庭、先辈等）的某些类型的数据的感觉。接着，用一条线把所学的东西串起来，直到最后以一个用到这些数据结构的实用软件结束。

5.6.7 小结

本节介绍了数据可以由多个简单论域构成，也就是说数据可以用算符代表的复合论域。算符中参数的位置必须与它们所表示的逻辑严格一致。算符不仅能以简单论域作为参数，也可以把其他算符作为参数。算符所表示的数据可以像其他 Prolog 变量一样使用，甚至可以执行所有操作，包括递归。

必须花费时间为正在开发的软件建立一些真实世界的模型子集，以获得对数据的感觉。同时应该用模型化的数据对所要用到的谓词进行测试，以期精益求精。

本章小结

本章介绍了关于 Prolog 编程最基本的内容，包括 Horn 子句、Prolog 推理机、程序控制、Prolog 算符等。这些内容也是 Visual Prolog 的基础知识，但在学习时应注意两者之间

的差异。

习题 5

- 1. Horn 子句指的是什么？何谓 Horn 子句逻辑？
- 2. Horn 子句与 Prolog 程序有什么关系？
- 3. 首先装入 Visual Prolog 6 随软件携带的例子，然后用 Prolog 推理机（PIE）测试。
- 4. 假设在 Prolog 里已经包含了有关“积木世界”的下列事实与规则：

```
on(b,a).
on(c,b).
on(f,c).
on(c,d).
three_store(T,M,B):- on(T,M), on(M,B).
four_store(A,B,C,D):- three_store(A,B,C), three_store(B,C,D).
```

现在给出目标：

```
?- three_store(Fst,Snd,Trd).
```

给出其求解过程。并通过其求解过程掌握 Prolog 的执行策略，包括“回溯”、“匹配”、“失败”的含义。

- 5. 某一人数众多的俱乐部的成员们被分成 3 组，每组队员只能向同组或本组下面的组中的成员挑战（如果存在）。

写出一个 Visual Prolog 程序显示这种比赛方式下所有可能的比赛：

```
tom versus bill
marjory versus annette
```

例如，使用截断确保

```
tom versus bill
```

和

```
bill versus tom
```

不能被同时显示。

- 6. 编写一个尾部递归程序，输出 2 的幂值表，如：

```
N      2^N
--      ----
1         2
2         4
3         8
```

4	16
M	M
10	1024

如上显示，使它在 10 次幂结束。

7. 编写一个尾部递归程序，使之可以接受一个数字作为输入，并可以用两种方法之一结束。它将通过自身数字反复相乘，直到 81 或大于 100 的数为止。如果数值到达 81，它将输出 yes。如果数值超过 100，它将输出 no。

第6章 类与对象

本章主要介绍在 Visual Prolog 6 中的面向对象的概念，并将列举一些实例，以使读者尽快熟悉这个概念。主要内容包括对象模型、类实体、模块、创建和访问对象、接口对象类型、多重实现、包容多态性、Support 类型扩展、Object 超类型、继承以及 Visual Prolog 5 与 Visual Prolog 6 的差别等。

6.1 对象模型

在 Visual Prolog 6 中，对象模型的语义实体是对象、对象类型和类。有关这些实体的语法概念是接口、类的声明及实现。接口是一组命名的谓词声明。接口描述了对象之间的“界面”，并因此而得名，即它是从一个对象之外进入到对象内部的入口。接口描述了对象类型。

考虑以下接口定义：

```
interface person
  predicates
    getName : () -> string Name.
    setName : (string Name).
end interface person
```

这是一个名为 `person` 的接口定义。在这里，所有 `person` 类的对象都有两个谓词，即 `getName` 和 `setName`，其声明如上所示。

接口只定义对象的类型；而对象由类产生。一个类包含类的声明和类的实现。一个创建 `person` 对象的类可以这样声明：

```
class person_class : person
  constructors
    new : (string Name).
end class person_class
```

这是一个名为 `person_class` 的类的声明，可以由 `person_class` 类构造 `person` 类型的对象。这个类有一个名为 `new` 的构造函数，给 `new` 一个 `Name` 就能创建一个对象（属于 `person` 类型）。

这个类还需要一个实现，如下面的代码所示：

```
implement person_class
  facts
    name : string.
```

```

clauses
  new(Name) :- name := Name.

clauses
  getName() = name.

clauses
  setName(Name) :- name := Name.
end implement person_class

```

这是名为 `person_class` 的类实现。这个实现必须为诸如 `new`, `getName` 和 `setName` 的每个公共谓词和构造函数提供定义。这个实现也可以局部声明和定义附加实体，这种实体仅在这个实现中可见。在这个例子中，这个类声明了一个名为 `name` 的事实变量以存储人的名字。

对于事实变量 `name`，每个对象都有自己的实例。上面子句中的代码都能引用具有事实变量的那个特例。通常把这种谓词称为对象谓词，把这个事实变量称为对象事实。

6.2 类实体

一个类也可以有为这个类的所有对象共享的实体。例如，将上面的例子扩展，添加代码来为 `person_class` 类创建的对象个数计数。这个数字会随着每个对象的创建而增加，且永不减少。

用一个谓词，即谓词 `getCreateCount`，将类声明予以扩展，这个谓词用于返回当前计数值：

```

class person_class : person
  constructors
    new : (string Name).
  predicates
    getCreatedCount : () -> unsigned Count.
end class person_class

```

注意：公共可访问类谓词在类声明中进行声明，而公共可访问对象谓词在接口中声明。这个规则没有例外。不可能在类声明中声明对象谓词，也不可能在一个接口中声明类谓词。

这个谓词需要在类实现中定义。此外，还需要一个事实来存储计数值。这个事实必须是一个类事实，即它为所有对象所共享。在一个类的实现里，可以声明并定义私有对象实体以及私有类实体。声明类实体时，在相关的声明段前加关键词 `class`。

`person_class` 类的实现如下：

```

implement person_class
  class facts

```

```

        createdCount : unsigned := 0.

    clauses
        getCreatedCount() = createdCount.

    facts
        name : string.

    clauses
        new(Name):-
            name := Name,
            createdCount:= createdCount+1.

    clauses
        getName() = name.

    clauses
        setName(Name):- name := Name.
end implement person_class

```

在本例中，添加了一个类事实 `createCount`，并将其初始化为零。并且为谓词 `getCreateCount` 添加了一个子句，用于返回事实 `createdCount` 的当前值。最后，在构造函数中添加代码，使变量 `createdCount` 递增。

注意：在构造函数中，有两处赋值形式相同，但一个是更新对象状态的，另一个是更新类的状态的。

6.3 模块

类的一种特殊变体，根本不能产生对象，所以就它们所起的作用来讲，应称之为“模块”，而不是类。一个非构造的对象类（或者直接称为模块）在声明中略去对象的类型：

```

class io % no type here
    predicates
        write : (string ToWrite).
        write : (unsigned ToWrite).
end class io

```

这样一个不能创建对象的类很明显是不能包含对象实体的，也不可能构造函数。

6.4 创建和访问对象

基于上述代码，可以创建一个能创建对象且用 `io` 类来写人名的目标（goal）（在此对

io 的类实现暂时不做考虑)。

```
goal
    P = person_class::new("John"),
    Name = P::getName(),
    io::write(Name).
```

第1行, 调用 `person_class` 类构造函数 `new`。创建的对象绑定到变量 `P`。第2行, 引用了 `P` 中对象谓词 `getName`, 并将结果绑定到变量 `Name`。最后一行调用了 `io` 类的类谓词 `write`。

注意: 引用类中的名字时要用双冒号, 如 `person_class::new`。同样, 引用对象谓词时要用单冒号, 如 `P::getName`。

尽管构造函数并不像一般函数那样声明, 但它们是返回对象的函数: 返回类型包含在类声明中。

6.5 接口对象类型

前面已经提到, 接口是对象类型。按照字面意义来讲, 在用到非对象类型的地方是可以使用接口的。例如, 在如下谓词声明中:

```
class mail
    predicates
        sendMessage : (person Recipient, string Message).
end class mail
```

谓词 `mail::sendMessage` 以 `person` (接口) 和 `string` 作为参数。

6.6 多重实现

可以创建多个完全不同的类, 这些类都创建 `person` 对象。只需声明和实现更多的可构造 `person` 对象的类。这些类的实现可以有很大区别, 比如, 可以创建一个将 `person` 存放在数据库中的类。以下便是这样一个类的声明:

```
class personInDB_class : person
    constructors
        new : (string DatabaseName, string Name).
end class personInDB_class
```

这里不关心具体的实现。下面的代码显示的是一个关于某特定对象类型的对象, 但它可以有完全不同的实现。

```
implement personInDB_class
    facts
```



```

    db : database.
    personID : unsigned.

    clauses
        new(DatabaseName, Name):-
            db := database_class::getDB(DatabaseName),
            personID := db:storePerson(Name).

    clauses
        getName() = db:getPersonName(personID).

    clauses
        setName(Name) :- db:setPersonName(personID, Name).
end implement personInDB_class

```

值得注意的是，不仅内部行为完全不同，在内部状态上，结构和内容也全然不同。

6.7 包容多态性

无论同一种类型的对象的实现有多大区别，它们都可以用在同一个场合里。例如，可以用上面所定义的 `mail` 类向一个人（对象）发送消息，不管那个人是由 `person_class` 还是 `personInDB_class` 构造的。

```

goal
    P1 = person_class::new("John"),
    mail::sendMessage(P1, "Hi John, ..."),
    P2 = personInDB_class::new("Paul"),
    mail::sendMessage(P2, "Hi Paul, ...").

```

这种行为称为包容（subsumption）：只要两个对象都是那段上下文需要的类型，那么由这个类构造的对象或由那个类构造的对象一样可用。

还可以看到，谓词 `mail::sendMessage` 可以用于任意 `person` 类的对象中，所以从某种意义上讲，即从包容多态（subsumption polymorphism）意义上讲，这个谓词是多态的（polymorphic）。

6.8 support 类型扩展

可以想像，在程序中会涉及一种特殊的人，即程序的用户。用户有名字，还有一个密码。想为用户创建一个新的对象类型或接口，以规定用户是一个人，且有一个密码。为此，使用支持限定符（support qualification）：

```

interface user supports person
  predicates
    trySetPassword : (string Old, string New, string Confirm) determ.
    validatePassword : (string Password) determ.
end interface user

```

上面代码中，规定了 `user` 支持 `person`。这有两个作用：

(1) 它意味着 `user` 对象必须提供在 `person` 接口中所声明的谓词（如谓词 `getName` 和 `setName`）。

(2) `user` 类型的对象同时也是 `person` 型的对象，因此也能用于期望使用 `person` 对象的上下文中。

也就是说，假设有一个 `user` 类：

```

class user_class : user
  constructors
    new : (string Name, string Password).
end class user_class

```

那么，这个类的对象就可以被 `mail::sendMessage` 使用：

```

goal
  P = user_class::new("Benny", "MyCatBobby"),
  mail::sendMessage(P, "Hi Benny, ...").

```

一个接口可以支持多个其他的接口，也就是说：

- 该种类型的对象必须提供在被支持的接口中的全部谓词；
- 该种类型的对象同样具有所有其他的类型。

一个接口同样可以支持一个或多个接口，而这些接口本身也能支持一个或多个接口，如此等等。同样在这个例子中：

- 该种类型的对象必须提供被间接及直接支持的接口中的所有谓词；
- 该种类型的对象具有所有其他的间接类型及直接类型。

这种支持限定产生了子类型层次，称 `user` 为 `person` 的一个子类型。

6.9 object 超类型

一个接口并不明确地支持其他任何接口，但却隐含地支持 `object` 接口。`object` 是一种隐含定义的、没有内容（即没有谓词）的接口。任何对象都直接或间接地支持 `object` 接口，所以任何对象都含有 `object` 类型。因此，称 `object` 为所有对象类型的超类型。

6.10 继承

当对 `user_class` 类进行类实现时，当然可以利用 `person` 类中的代码来减少工作量。假

设 `user` 类与 `person_class` 类很相似，不同之处仅在于 `user_class` 类还涉及密码。想要 `user_class` 继承 `person_class` 类中 `person` 的部分实现，可以用下面的代码完成：

```
implement user_class
  inherits
    person_class

  facts
    password : string.

  clauses
    new(Name, Password):-
      person_class::new(Name),
      password := Password.

  clauses
    trySetPassword(Old, New, Confirm):-
      validatePassword(Old),
      New = Confirm,
      password := New.

  clauses
    validatePassword(Password):-
      password = Password.
end implement user_class
```

这个实验表明它继承了 `person_class`，这样会有如下作用：

- 一个 `person_class` 类的对象被植入到每个构造的 `user_class` 对象中；
- 接口 `person` 中的所有谓词可直接从 `person_class` 继承到 `user_class` 中。

当继承一个谓词时，不再需要直接描述它的实现，而是使用所继承类中的谓词实现。

在某种意义上，继承可以看作是语法上的修饰。至少还可以通过下面的代码达到相同的效果（有关密码谓词的子句同上）：

```
implement user_class
  facts
    person : person.
    password : string.

  clauses
    new(Name, Password):-
      person := person_class::new(Name),
      password := Password.

  clauses
    getName() = person:getName().
```

```
clauses
    setName(Name) :- person:setName(Name).

...

end implement user_class
```

在这段代码里，没有继承 `person_class` 类，而是创建了一个 `person_class` 类的对象并将其存储到一个事实变量中。这里没有继承 `getName` 和 `setName` 的代码，而是再次对这两个谓词做了一些琐碎的实现，它们直接将相应的任务委托给事实变量中的对象。

这一段代码与前面的代码有着几乎相同的作用，但是毕竟还有一些显著的区别：首先，需要写更多的代码。其次，`person_class` 类没有被植入 `user_class` 类之中，相反有一个对它的引用。而且，在此涉及两次内存分配，而不是一次。最后，可以动态地将事实变量的值改变为另一个对象，这只需给事实变量指派一个新的对象即可实现。例如，将其改变成 `personInDB_class` 类的一个对象。

应当注意，第2个实现里有一个间接的调用。`Visual Prolog` 处理这种间接调用时很有效，但在处理继承时更加高效。

`Visual Prolog 6` 支持多重继承，即可以同时从多个类继承。

6.11 对象体系的其他特点

以上介绍了 `Visual Prolog 6` 的对象体系里的最为基础的概念。在对象体系中还有别的一些有趣的特点，在此不做介绍，例如：

- 对象支持实现中的更多的接口。
- 存储器回收生效的确定者(`finalizer`)。
- 可以与 C# 中的委派无缝配对的对象谓词值。

6.12 Visual Prolog 5 与 Visual Prolog 6 的差异

本节主要介绍 `Visual Prolog 6` 和 `Visual Prolog 5` 之间的差异。焦点集中在 `Visual Prolog 5` 已经被改变了的特性，而不是新增加的特性。

6.12.1 句点

句点(dots)，所有的声明（即常量、论域、谓词和事实）均以句点（即“.”）终止。

6.12.2 谓词

谓词(predicates)声明的语法在各方面已经被改变。这个例子说明了其大部分的改变。

predicates

```
ppp : (integer Input) procedure (i).
qqq : (integer Input) -> integer Output procedure (i).
```

在谓词名字(在声明中总是第1个字)和谓词类型之间有一个冒号。

返回类型已经被移到箭头(即“->”)之后的类型表达式的尾部。

谓词的类型、样式和流模式之间没有破折号。

声明前面的谓词样式格式不再存在。

如果谓词在一个类或接口声明中进行声明,则所省略的谓词流模式意味着所有的参数均为输入参数。

6.12.3 谓词论域

谓词论域(predicate domains)声明的语法同样已经被改变。

domains

```
pppDom = (integer Input) procedure (i).
qqqDom = (integer Input) -> integer Output procedure (i).
```

6.12.4 引用论域

引用论域(reference domains)不能引用一个非引用型的用户论域。

下面的 Visual Prolog 5 代码:

domains

```
xref = reference xref(y)
y = integer
```

对应于下面的 Visual Prolog 6 代码:

domains

```
xref = reference xref(yref).
yref = reference integer.
```

6.12.5 函数子句

函数子句(function clauses)的语法已经改变,因此返回值被放在子句头的等号之后。

clauses

```
qqq(X) = 1 :-
    X < 3,
    !.
qqq(X) = X.
```

6.12.6 常量

常量 (constants) 不再是宏，但必须是一个确定类型的值。

constants

```
myList: integer_list = [14, 56, -3].
```

对数字常量、字符常量和串常量而言，类型可以被忽略。

6.12.7 事实

事实 (facts) 声明类似谓词声明：

facts - myFactDB

```
fact1 : (integer Input) nondeterm.
```

冒号以及样式声明 (即 `nondeterm`) 写在尾部。事实仍然可以为 `single`, `determ` 和 `nondeterm`。

事实段只能用在一個类的实现内部。

事实修饰符 `nocopy` 不再使用。所有事实都作为 `nocopy` 方式处理。

事实段以关键字 `facts` 标识。可选择的 `database` 不再是一个关键字。

6.12.8 事实变量

在 Visual Prolog 6 中，事实变量 (fact variables) 是一个新概念。它们在事实段进行声明：

facts

```
myFactVariable : integer_list := [14, 56, -3].
```

Visual Prolog 5 的用户认为这等价于下面伴随有初始化的单个事实：

facts

```
myFact : (integer_list Value) single.
```

clauses

```
myFact([14, 56, -3]).
```

事实变量用起来像变量一样，因为它来自于命令性 (imperative) 语言：

clauses

```
p() :-
    myFactVariable := [14, 56, -3],
    q(myFactVariable).
```

Visual Prolog 5 的用户认为这等价于下面的代码：

```

clauses
    p() :-
        assert(myFact( [14, 56, -3])),
        myFact(Value), q(Value).

```

6.12.9 嵌套表达式与函数

嵌套的表达式与函数 (nested expressions and functions) 实际上在任何地方都可以嵌套。

```

clauses
    factorial(0) = 1 :-
        !.
    factorial(N) = N * factorial(N-1).

```

从上面的代码可以看到，第2个子句的返回值是一个表达式。这个表达式包含一个函数调用，并且给这个函数的参数也是一个表达式。

6.12.10 编译器命令

编译器命令 (compiler directives) 以字符 “#” 开始。例如：

```
#include @"packageAaa\packageAaa.ph"
```

一串文字前面的符号 “@” 意味着转义序列不被使用，即 @"\n" 代表两个符号 “\” 和 “n”。

6.12.11 条件编译

条件编译 (conditional compilation) 只适用于 Visual Prolog 6 中的段 (sections)。例如：

```

#if a::myconst = 1 #then
clauses
    p(0) = 1.
#endif

```

这段 Visual Prolog 5 代码：

```

clauses
    p(X):-
        ifdef debug_mode
        write("X",X),
        endif
        q(X).

```

对应于下面的 Visual Prolog 6 代码。

(1) 编译时间常量 `debug_mode` 应当放在某个类中。假设命名它为全局常量 `globalConstants`。

(2) 额外需要一个谓词来表示该子句的一部分，这部分可能位于条件编译之内。

```
class predicates
    writeX : (integer X).
#if globalConstants::debug_mode = 1 #then
    clauses
        writeX(X):-
            stdIO::write("X=",X).
    #else
    clauses
        writeX(_).
    #endif
    clauses
        p(X):-
            writeX(X),
            q(X).
```

6.12.12 输入输出及特殊论域

特殊论域指 `file` 论域与 `db_selector` 论域。`file` 论域放在 `fileSelector` 类中。`db_selector` 论域放在 `chainDBSelector` 类中。文件 `pfc\5xVIP\fileSelector.cl` 和文件 `pfc\ChainDB\chainDBSelector.cl` 应当在它们被修改之前复制到项目目录（适当的子目录）。

引入论域只考虑向后的兼容性。新的风格是使用对象进行代替。

在 Visual Prolog 6 中，IO 处理是基于流的。流谓词使用匿名（anonymous）参数类型和省略（ellipsis）符号。

6.12.13 省略与匿名参数类型

如果许多参数是可变的，则可以使用省略号（ellipsis）。省略号是一个特殊的符号，代表“任意类型的零个或多个参数”。其语法如下：

```
class predicates
    p : (...).
    clauses
        p(...):-
            stdio::write(...).
```

如果一个参数的类型在编译时间是未知的，则可以使用匿名（anonymous）参数。其语法如下：


```
class predicates
    setProperty : (_).
```

6.12.14 对象与类

Visual Prolog 5 与 Visual Prolog 6 对象模型之间的差别已在前面叙述了。

6.12.15 库支持

库支持叙述 Visual Prolog 6 与 Visual Prolog 5 之间名字的等价性。

转换表 6.1 描述了 Visual Prolog 6 与 Visual Prolog 5 之间的等价名字。

表 6.1 Visual Prolog 6 与 Visual Prolog 5 之间名字的等价性

Visual Prolog 5 名字	Visual Prolog 6 名字
database	facts
include	#include
elseif	#else
endif	#endif
abs	math::abs
exp	math::exp
ln	math::ln
log	math::log
sqrt	math::sqrt
round	math::round
trunc	math::trunc
cos	math::cos
sin	math::sin
tan	math::tan
arctan	math::arctan
random	platformSupport5x::random
randominit	math::randomInit
true	succeed
cutbacktrack	programControl::cutBackTrack
getbacktrack	platformSupport5x::getBackTrack
binary_to_strlist	conversion5x::binary_to_strlist
upper_lower	conversion5x::upper_lower
char_int	conversion5x::char_int
real_ints	conversion5x::real_ints
str_char	conversion5x::str_char
str_dosstr	conversion5x::str_dosstr
str_int	conversion5x::str_int
str_real	conversion5x::str_real

续表

Visual Prolog 5 名字	Visual Prolog 6 名字
str_ref	conversion5x::str_ref
compressbinary	binary::compress
expandbinary	binary::expand
crc32binary	binary::calculateCRC
errorexit	breakControl5x::errorExit
exit	breakControl5x::exit
errmsg	breakControl5x::errmsg
lasterror	breakControl5x::lasterror
ptr_dword	platformSupport5x::ptr_dword
bitand	platformSupport5x::bitand
bitleft	platformSupport5x::bitleft
bitnot	platformSupport5x::bitnot
bitor	platformSupport5x::bitor
bitright	platformSupport5x::bitright
bitxor	platformSupport5x::bitxor
membyte	platformSupport5x::membyte
memdword	platformSupport5x::memdword
memword	platformSupport5x::memword
sound	platformSupport5x::sound
sleep	platformSupport5x::sleep
beep	platformSupport5x::beep
date	platformSupport5x::date
difftime	platformSupport5x::difftime
marktime	platformSupport5x::marktime
timeout	platformSupport5x::timeout
time	platformSupport5x::time
storage	platformSupport5x::storage
comline	platformSupport5x::comline
concat	string5x::concat
format	string5x::format
frontchar	string5x::frontchar
frontstr	string5x::frontstr
fronttoken	string5x::fronttoken
isname	string5x::isname
searchchar	string5x::searchchar
searchstring	string5x::searchstring
str_len	string5x::str_len
subchar	string5x::subchar
substring	string5x::substring
asciiz_2_vb_string	string5x::asciiz_2_VB_String
list_to_string	string5x::list_to_string

续表

Visual Prolog 5 名字	Visual Prolog 6 名字
separate_string	string5x::separate_string
str_strcmp	string5x::str_StrCmp
str_strcmpi	string5x::str_StrCmpi
str_strncmp	string5x::str_StrNCmp
file_time	platformSupport5x::file_time
mem_SystemFreeGStack	memory::systemFreeGStack
save	file5x::save
consult	file5x::consult
bt_close	chainDb8::bt_close
bt_copyselector	chainDb8::bt_copyselector
bt_create	chainDb8::bt_create
bt_delete	chainDb8::bt_delete
bt_open	chainDb8::bt_open
bt_statistics	chainDb8::bt_statistics
bt_updated	chainDb8::bt_updated
chain_delete	chainDb8::chain_delete
chain_first	chainDb8::chain_first
chain_last	chainDb8::chain_last
chain_next	chainDb8::chain_next
chain_prev	chainDb8::chain_prev
db_begintransaction	chainDb8::db_begintransaction
db_btrees	chainDb8::db_btrees
db_chains	chainDb8::db_chains
db_close	chainDb8::db_close
db_copy	chainDb8::db_copy
db_create	chainDb8::db_create
db_delete	chainDb8::db_delete
db_endtransaction	chainDb8::db_endtransaction
db_flush	chainDb8::db_flush
db_garbagecollect	chainDb8::db_garbagecollect
db_open	chainDb8::db_open
db_openinvalid	chainDb8::db_openinvalid
db_reuserefs	chainDb8::db_reuserefs
db_setretry	chainDb8::db_setretry
db_statistics	chainDb8::db_statistics
db_updated	chainDb8::db_updated
key_current	chainDb8::key_current
key_delete	chainDb8::key_delete
key_first	chainDb8::key_first
key_insert	chainDb8::key_insert
key_last	chainDb8::key_last

续表

Visual Prolog 5 名字	Visual Prolog 6 名字
key_next	chainDb8::key_next
key_prev	chainDb8::key_prev
key_search	chainDb8::key_search
term_delete	chainDb8::term_delete
closefile	file5x::closefile
eof	file5x::eof
filemode	file5x::filemode
filepos	file5x::filepos
flush	file5x::flush
openappend	file5x::openappend
openfile	file5x::openfile
openmodify	file5x::openmodify
openread	file5x::openread
openwrite	file5x::openwrite
readdevice	file5x::readdevice
writedevice	file5x::writedevice
copyfile	file5x::copyfile
deletefile	file5x::deletefile
existfile	file5x::existfile
fileattrib	file5x::fileattrib
renamefile	file5x::renamefile
searchfile	file5x::searchfile
filenameext	file5x::filenameext
filenamepath	file5x::filenamepath
filenamereduce	file5x::filenamereduce
filenameunique	file5x::filenameunique
dirclose	file5x::dirclose
dirfiles	file5x::dirfiles
dirmatch	file5x::dirmatch
diropen	file5x::diropen
disk	file5x::disk
diskspace	file5x::diskspace
mkdir	file5x::mkdir
rmdir	file5x::rmdir
readblock	file5x::readblock
file_bin	file5x::file_bin
file_str	file5x::file_str
readchar	file5x::readchar
readint	file5x::readint
readln	file5x::readln
readreal	file5x::readreal

续表

Visual Prolog 5 名字	Visual Prolog 6 名字
nl	file5x::nl
write	file5x::write
writeln	file5x::writeln
writeblock	file5x::writeblock
getbinarysize	binary::getSize
makebinary	binary::create
getbyteentry	binary::getIndexed_unsigned8
getwordentry	binary::getIndexed_unsigned16
getdwordentry	binary::getIndexed_unsigned32
getrealentry	binary::getIndexed_real
setbyteentry	binary::setIndexed_unsigned8
setwordentry	binary::setIndexed_unsigned16
setdwordentry	binary::setIndexed_unsigned32
setrealentry	binary::setIndexed_real
envsymbol	platformSupport5x::envsymbol
osversion	platformSupport5x::osversion
syspath	platformSupport5x::syspath
system	platformSupport5x::system
cast	uncheckedConvert
val	convert

一般来说，库是按照对象概念新编写的。有一个特殊的库 5xVip，包括子文件夹。该库的目的是用来移植 Visual Prolog 5 代码。新的程序不应该使用 5xVip 库。

本章小结

本章主要介绍了 Visual Prolog 6 中的对象模型、类实体、模块、创建和访问对象、接口对象类型、包容多态性、Support 类型扩展、Object 超类型、继承，以及 Visual Prolog 5 与 Visual Prolog 6 的差别等。这些内容是 Visual Prolog 6 的面向对象的基本概念。本章通过列举一些实例，使读者尽快熟悉这些概念。

习题 6

1. 解释 Visual Prolog 6 中的对象模型、类实体、模块等基本概念及其含义。
2. 在 Visual Prolog 6 中，如何创建和访问对象？何谓接口对象类型？
3. 何谓“多重实现”？其含义是什么？
4. 什么叫“包容多态性”？其含义是什么？

-
5. 什么叫“继承”？其含义是什么？
 6. 何谓 Support 类型扩展、Object 超类型？
 7. Visual Prolog 的对象体系与其他语言有何区别与联系？
 8. Visual Prolog 5 与 Visual Prolog 6 有何差异？
 9. 如何将 Visual Prolog 5 的程序移植到 Visual Prolog 6？

第 7 章 Visual Prolog 编程

本章介绍基于 Visual Prolog 编程方面的知识，主要内容包括 Visual Prolog 基础、Visual Prolog 的 GUI 编程、Visual Prolog 的逻辑层和 Visual Prolog 的数据层。

7.1 Visual Prolog 基础

传统的 Prolog 与 Visual Prolog 6 之间的差别主要体现在如下几个方面：

(1) 程序结构

很明显，传统 Prolog 中所使用的结构与 Visual Prolog 6 中使用的结构，在理解的难易程度方面不同。主要包括如何规划来自定义（definitions）的声明（declarations），以及如何简要地说明程序必须使用指定关键字（keywords）进行查找的主目标。

(2) 文件考虑

Visual Prolog 6 提供了各种工具，以便将程序结构组织成不同类型的文件。

(3) 作用域访问

Visual Prolog 6 可以挑选在其他模块中通过使用称为作用域标识（scope identification）的概念而开发出来的功能。

(4) 面向对象

Visual Prolog 6 程序可以编写面向对象的程序，使用标准的面向对象特性。

7.1.1 程序结构

Visual Prolog 的程序从结构上讲，主要包括若干个段，即论域段、谓词段、子句段、目标段等。Visual Prolog 作为强类型的编译型语言，通常用论域段和谓词段来给出有关的声明或定义。

1. 声明与定义

声明（declaration）与定义（definition）有着不同的含义。

在 Prolog 中，当需要使用一个谓词的时候，就可以直接使用，无需事先向 Prolog 推理机做任何的公告。例如，在前面的例子中，grandFather 谓词的子句就是利用传统的 Prolog 的谓词头和谓词体结构直接写的。不用在代码中再告知推理机这个谓词结构是后面需要使用的。

类似地，当在传统的 Prolog 中使用一个复合论域时，无需首先告诉 Prolog 推理机关于使用该论域有何意图。只要感到需要，就可以直接使用一个论域。

然而，在 Visual Prolog 6 中，在编写一个谓词的子句体代码之前，必须首先对编译器

声明这样一个谓词的存在。类似地，在使用任何论域之前也必须先进行声明，以便将这些论域的存在告知编译器。

在 Visual Prolog 6 中需要这种预先告知功能的原因本质上是为了保证将运行时间异常 (running exceptions) 尽可能地转变为编译时间错误 (compile time errors)。

对于“运行时间异常”，指的是只在运行所编译的程序期间出现的问题。例如，如果想使用一个整数作为一个函数的参数，但是却错误地使用了实数，这就会成为一个运行错误（这大都出现在使用其他编译器的程序中，但不是在 Visual Prolog 6 中），程序就会因此而失败。

当声明已定义的谓词和论域时，这类位置语法，即哪个参变量属于哪个论域，就会对编译器起作用。因此，当 Visual Prolog 6 执行编译时，它将比较彻底地检查程序，以发现诸如此类的语法错误及其他错误。

由于 Visual Prolog 6 的这些特性，整个程序的效率因此提高了。程序员不必等到程序实际执行时才发现错误。事实上，对于实际编写程序的人，将体会到这大大地节约了时间。通常，运行时导致发生运行时间异常的条件如此难以捉摸，以至于错误可能会在很多年后才被发现，或者会在许多特别重要的情况或令人尴尬的场合表现出来。

所有这些表明，编码中存在的论域和谓词要在定义前给出合适的声明，以给编译器详尽的指示。

2. 关键字

一个 Visual Prolog 6 的程序包括一组被标点分为不同部分的代码，由特定的关键字告诉编译程序所要生成的类型。例如，关键字可以将谓词和论域的定义和声明区分开。通常，每一部分由一关键字开始，在每一部分结束时，一般没有关键字指示。新的关键字的出现表明前一部分的结束和下一部分的开始。

对这一规则的例外是关键字 **implement** 和 **end implement**，在这两个关键字中间的代码表示它们属于一个特殊的类。若有人不懂类的概念，可以把它看作程序的一个模块或一个部分。

本节将只介绍下述关键字。同样给出了这些关键字的用途，具体的句法可以从文档资料中学到。Visual Prolog 6 中还有其他一些关键字，可以在以后的内容和文档资料中学到。

本章需要掌握的关键字在下面分别描述。

implement 和 **end implement**

在这里讨论的所有关键字中，这是惟一成对存在的。Visual Prolog 6 把出现在这两个关键字之间的代码看成属于一个类。这个类名必须在 **implement** 这个关键字后给出。

open

这个关键字用来扩展类的作用域的可见性，它被用在 **implement** 这个关键字之后。

constants

这个关键字用来表明一部分经常在程序中被使用的代码。例如，如果字符串“PDC

Prolog”在程序中多次出现，那么就可以定义一个它的缩写：

```
constants  
    pdc="PDC Prolog".
```

注意：常量的定义以句号结束。与 Prolog 中的变量不同，常量应该以小写字母开头。

domains

这个关键字用来标明程序中将要用到的论域。这种论域声明的句法中有许多变量的声明，用来指示许多将来在程序中要用到的论域。因为本节介绍 Visual Prolog 基础性内容，将不对论域的具体细节进行讨论。

总结一下，这里将声明那些用于论域的算符和构成算符变元的论域，算符和复合论域在本书的前面章节部分有详细解释。

class facts

这个关键字指定一个段，这个段用来声明将在程序代码中出现的事实。每个事实由一个符号化的名字声明每个事实的变元和各个变元所属的论域。

class predicates

这一段将包含在子句部分被定义的谓词的声明。同样，谓词的名称以及变元和论域也在这一段中被声明。

clauses

在 Visual Prolog 6 代码的所有部分中，这一部分和传统的 Prolog 程序最为相似，它包含对已声明谓词的定义，会发现在这里使用的谓词与 **class predicates** 部分中声明的谓词句法相同。

goal

这一段定义是 Visual Prolog 6 程序的主要入口点。更详细地解释将在下面给出。

7.1.2 目标

在传统的 Prolog 中，只要谓词在代码中被定义了，Prolog 核心程序就会被引导从那个谓词开始程序的执行。但是，在 Visual Prolog 6 中不是这样，作为一个编译程序，它要生成高效率的程序执行代码。在编译程序工作的时候，代码事实上并未被执行。所以编译程序需要事先知道程序从哪个谓词开始执行，这样当程序被调用执行时，它就能从正确的地方开始。正如所期望的那样，这个编译好的程序可以不再需要 Visual Prolog 编译程序和 VDE 而独立运行。

为了实现这些功能，有一个专门由关键字 **goal** 指示的段。把它们作为没有自变量的特殊谓词考虑，这种谓词就是程序开始执行的地方。

7.1.3 文件考虑

通常，将程序的所有部分放在一个文件里是很麻烦的，这样会使程序难于理解，甚至有时会产生错误。Visual Prolog 6 使用 VDE 可以将程序代码分成不同的文件，也可使用 VDE 将不同的代码写入不同的文件。借助这种方式，通过查找文件就可将经常用到的程序段找到。如果在许多文件中都要用到一个论域，那么可以在一个单独的文件中声明这个论域，然后这个文件可以被其他文件所访问。

然而，为了简化这个专门教程，应该主要使用一个文件写这些代码。在构造程序的过程中，VDE 可以自动生成更多当时可以忽略的程序，这将在以后的内容中学到。

7.1.4 作用域访问

Visual Prolog 6 将整个程序划分为不同的部分，每一部分定义一个类。在面向对象的程序语言中，类是一组程序代码和与之相关的数据的集合。这些内容在以后的叙述中将做更多的解释。像前面提到的一样，对于不熟悉面向对象程序语言的学习者，可以将类类似地考虑为模块。通常，Visual Prolog 6 在自己专门的文件中定义各个类。

在程序执行的过程中，程序经常需要调用在另一个类中定义的谓词。类似地，在一个类中定义的数据和论域可能需要允许能被另一个不同的文件所访问。

Visual Prolog 6 允许这些跨越类的代码数据引用，称为访问作用域。可以用一个例子来理解，假设在名为 class1 的类中定义了一个名为 pred1 的谓词（使用 VDE 在另一个文件中写出），在另一个文件 class2 中定义另一个名为 pred2 的谓词，下面就是如何在 pred2 的子句体中调用 pred1 的例子：

```
pred3:-
    ...
    !.

pred2:-
    class1::pred1, % pred1 is not known in this file.
                  % It is defined in some other file,
                  % Hence a class qualifier is needed
    pred3,
    ...
```

在上述例子中，可以看到 pred2 的子句体调用 pred1 和 pred3 这两个谓词，因为 pred1 在另一文件 class1 中被定义，因此将 class1 和“::”放在 pred1 的前面，这被称为是类的限定符。

但是谓词 pred3 和 pred2 一样，在相同的文件中被定义，因此没有必要在谓词前加上“class2::”来调用 pred3。

这种行为在专业上这样解释：pred3 的访问作用域蕴含于 pred2 中，因此没有必要澄清

pred3 和 pred2 一样来自于同一个类, 编译程序会在定义 class2 的范围内自动寻找 pred3 的定义。

某一特定类定义的作用域范围被限制在某一特定文件中声明的类中 (代码写在关键字 `implement` 和 `end implement` 之间), 在其中定义的谓词可以不用类名限定符和 “`::`” 符号作为前缀相互调用。

类的作用域范围可以通过使用 `open` 这个关键字予以扩充, 这个关键字可以通知编译程序调用在其他文件中定义的谓词、常量、论域名。如果作用域范围扩充了的话, 就不需要写类名限定符和 “`::`”。

```
open class1
...

pred3:-
    ...
    !.

pred2:-
    pred1,    % Note: "class1::" qualifier is not needed
              % anymore, as the scope area
              % is extended using the 'open' keyword
    pred3,
    ...
```

7.1.5 面向对象

Visual Prolog 的当前版本是一个强大的面向对象语言, 如果需要的话, 开发程序的整个代码会根据需要被放入合适的类中。即使对这种语言的面向对象特性不感兴趣, 它也会自动进行。在本章给出的例子中也发现这个特性。即使完全不使用任何该类生成的对象, 代码一样会被插入到名为 `family1` 的类中。这里将会直接使用这个类中的谓词代码。

本节不会涉及这种语言的面向对象特性, 将来的内容会拓展这个概念, 以便真正使用到面向对象的特性。

7.1.6 一个完整的例子: family1.prj6

可以把所学的知识集合起来去写第 1 个 Visual Prolog 6 的程序。这将包含 “Prolog 基础” 一章中所涉及的相同的基本逻辑。所有为本章编写的代码如下所示, 并写入名为 `family1.pro` 的文件中。使用 Visual Prolog 的 VDE 可完成代码的输入。本章所需要的更多文件将由 VDE 自动生成和维护。如何一步步使用 VDE 开发程序的步骤将在稍后介绍。

首先熟悉这个程序的主要代码。其内容如下:

```
implement family1
open core
```

constants

```
className = "family1".
classVersion = "$JustDate: $$Revision: $".
```

clauses

```
classInfo(className, classVersion).
```

domains

```
gender = female(); male().
```

class facts - familyDB

```
person : (string Name, gender Gender).
parent : (string Person, string Parent).
```

class predicates

```
father : (string Person, string Father) nondeterm anyflow.
```

clauses

```
father(Person, Father) :-
    parent(Person, Father),
    person(Father, male()).
```

class predicates

```
grandFather : (string Person, string GrandFather) nondeterm anyflow.
```

clauses

```
grandFather(Person, GrandFather) :-
    parent(Person, Parent),
    father(Parent, GrandFather).
```

class predicates

```
ancestor : (string Person, string Ancestor) nondeterm anyflow.
```

clauses

```
ancestor(Person, Ancestor) :-
    parent(Person, Ancestor).
ancestor(Person, Ancestor) :-
    parent(Person, P1),
    ancestor(P1, Ancestor).
```

class predicates

```
reconsult : (string FileName).
```

clauses

```
reconsult(FileName) :-
    retractAll(_, familyDB),
    file::consult(FileName, familyDB).
```

```

clauses
  run():-
    console::init(),
    stdIO::write("Load data\n"),
    reconsult("fa.txt"),
    stdIO::write("\nfather test\n"),
    father(X, Y),
    stdIO::writef("% is the father of %\n", Y, X),
    fail.
  run():-
    stdIO::write("\ngrandFather test\n"),
    grandFather(X, Y),
    stdIO::writef("% is the grandfather of %\n", Y, X),
    fail.
  run():-
    stdIO::write("\nancestor of Pam test\n"),
    X = "Pam",
    ancestor(X, Y),
    stdIO::writef("% is the ancestor of %\n", Y, X),
    fail.
  run():-
    stdIO::write("End of test\n").
end implement family1

goal
  mainExe::run(family1::run).

```

使用 VDE 开发程序的步骤简要概括如下:

(1) 在 VDE 中生成一个新的控制台程序

① 在启动 Visual Prolog 6 的 VDE 之后, 选择 Project→New, 如图 7.1 所示。

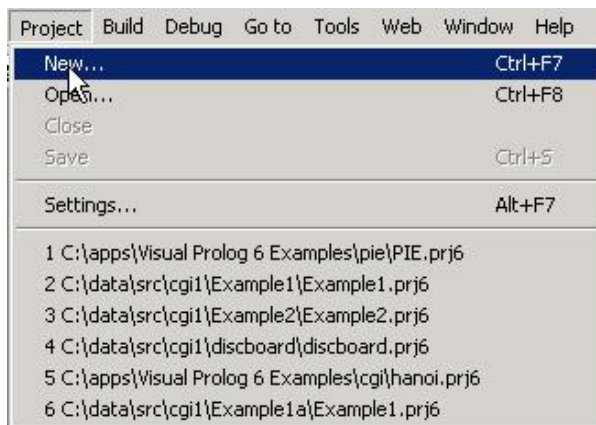


图 7.1 选择 New 菜单项

② 这时，一个对话框将被打开，如图 7.2 所示，输入所有相关信息，确定 UI Strategy 框中填写的是 Console，而不是 GUI。

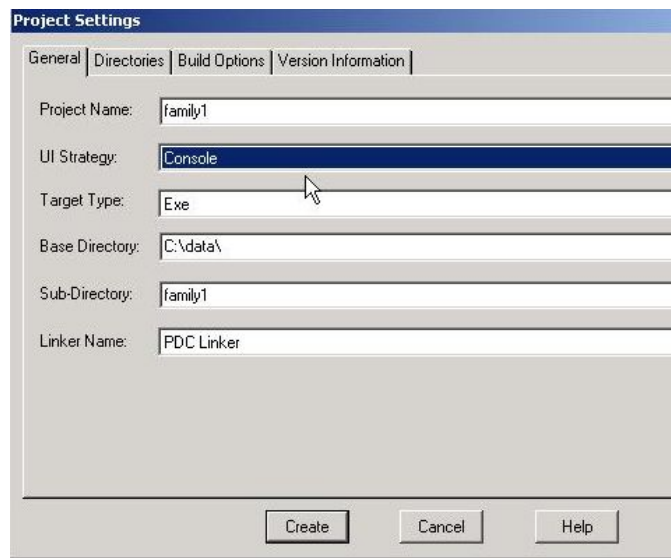


图 7.2 “项目设置”对话框

(2) 建立一个空的项目

① 当项目被建立时，VDE 将会显示如图 7.3 所示的项目窗口。此时，它还不知道这个项目依赖于哪个文件，它只是为这个项目构造出基本的源代码文件。

② 选择 Build→Build，如图 7.4 所示，编译和连接一个空的项目，以产生一个实际上不做任何事的可执行文件（这时也不会产生任何错误）。



图 7.3 项目窗口

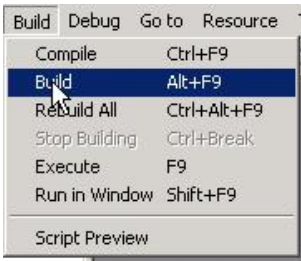


图 7.4 选择 Build 菜单项

当建立一个项目时，看看 VDE 的消息窗口中显示的消息（如果看不见消息窗口，可使用 VDE 中的 windows 菜单打开）。可以发现，VDE 已将所有相关的 Prolog 的基础类模块 PFC 放入该项目中，PFC 的类包含了建立程序所需的基本功能类。

(3) 使用实际的代码组装 family1.pro

① VDE 插入的是非常基本的代码，并不完成什么功能，如图 7.5 所示。可以删除整

个代码，只需要复制和粘贴本章中给出的 family1.pro 的代码到窗口中。

```
Copyright (c) Sabu Francis Associates

*****

implement family1
open core

constants
  className = "family1".
  classVersion = "",

clauses
  classInfo(className, classVersion).

clauses
  run():-
    console::init(),
    succeed(). % place your own code here
end implement family1

goal
  mainExe::run(family1::run).
```

图 7.5 VDE 生成的基本代码

② 当完成了复制和粘贴操作之后，family1.pro 的窗口将会如图 7.6 所示。

```
Copyright (c) Sabu Francis Associates

*****

implement family1
open core

| constants
  className = "family1".
  classVersion = "$JustDate: $$Revision: $".

clauses
  classInfo(className, classVersion).

domains
  gender = female(); male().

class facts - familyDB
  person : (string Name, gender Gender).
  parent : (string Person, string Parent).

class predicates
  father : (string Person, string Father) nondeterm anyflow;
```

图 7.6 复制和粘贴的 family1.pro 代码

(4) rebuild 代码

重新打开 **Build** 菜单, 若 **VDE** 发现源代码已经改变, 它将会重新编译改动的部分。如果选用 **Build All** 这一菜单项, 所有的模块都会被重新编译。对于大程序来说, 这将会耗费一定的时间。**Build All** 经常只是在一系列小的编译结束后使用, 只是为了保证每一步都按序进行。

在项目建立过程中, **VDE** 不只执行编译操作, 同样决定项目是否需要另外的 **PFC** 模块, 并插入必要的代码, 如果需要的话, 重新开始建立过程。这些可在消息窗口的消息中看到, 如图 7.7 所示, 在那里会看到由于一些附加的描述, **VDE** 要两次建立项目。

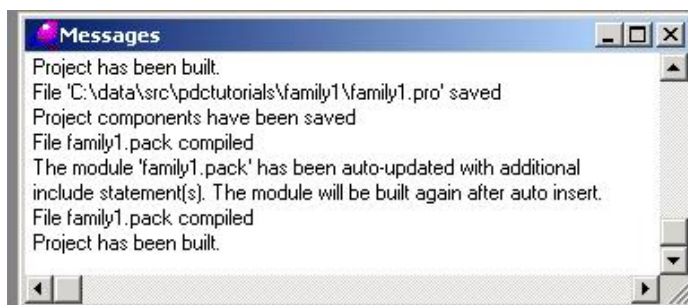


图 7.7 消息窗口

(5) 执行程序

现在完成了使用 **Visual Prolog 6** 编写的第 1 个应用程序。为了测试编译完的应用程序, 可以使用窗口菜单中的 **Build→Run** 命令。但是, 这样会导致错误。原因是小程序会为了函数功能去读取名为 **fa.txt** 的文本文件。这个文本文件包含程序运行所需要的数据, 以后将分析这个文本的语法。

现在需要使用文本编辑器来编写文本文件, 并把它们放置在可执行文件停留的路径上。通常, 可执行文件在主文件夹的名为 **exe** 的子文件夹中。

下面使用 **VDE** 来生成一个这样的文本文件。首先选择 **File→New**, 弹出创建项目的选项窗口 (**Create Project Item**), 从左边的列表中选取文本文件 (**text file**), 再选择文件可能停留的路径 (可执行文件 **family1.exe** 的路径)。然后给文件命名为 **fa.txt**, 再单击对话框上的 **Create** 按钮。直到文件名被给出, **Create** 按钮才会失效。最后确保检查框 **Add to project as module** 被选中。把该文件添加到项目中的好处是它将使后继的调试可以进行。

文件的内容如下:

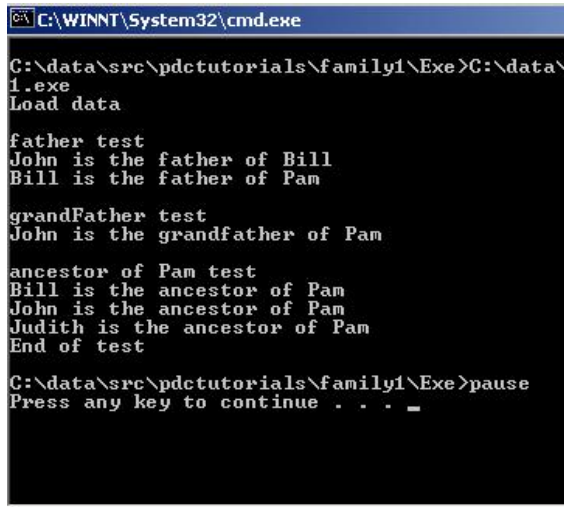
clauses

```
person("Judith",female()).
person("Bill",male()).
person("John",male()).
person("Pam",female()).
parent("John","Judith").
parent("Bill","John").
parent("Pam","Bill").
```


这是一个数据文件，注意到这个文件的语法非常相似于普通的 Prolog 代码。即使现在程序已经被编译结束并以二进制格式存在，也可以通过改变运行程序的主要数据来改变输出，采用与一般 Prolog 相同的语法，这些数据被写入文本文件。因此 Visual Prolog 6 仿效了传统 Prolog 动态代码的编写能力。虽然并不是所有的特性都可能，但至少在这个例子中的复合论域可以被给出。

文件 `fa.txt` 中使用的语法必须和程序的论域定义保持一致，定义人的算符必须用 `person` 来表示，否则用来表示算符的复合论域将不能得到初始化（在前面的叙述中已经提到过算符和复合论域）。

现在，使用热键 `Shift+F9` 来运行程序时（或使用窗口菜单中的 `Build→Run` 命令），将看到如图 7.8 所示的内容。



```
C:\WINNT\System32\cmd.exe

C:\data\src\pdctutorials\family1\Exe>C:\data\1.exe
Load data

father test
John is the father of Bill
Bill is the father of Pam

grandFather test
John is the grandfather of Pam

ancestor of Pam test
Bill is the ancestor of Pam
John is the ancestor of Pam
Judith is the ancestor of Pam
End of test

C:\data\src\pdctutorials\family1\Exe>pause
Press any key to continue . . . _
```

图 7.8 消息窗口

这个程序处理文件 `fa.txt` 中的信息，并运行所给人物的亲戚关系的逻辑顺序。

7.1.7 程序的取舍

程序的逻辑非常简单，这已经在前面讨论过，在那里解释了在一个 Prolog 程序中正确的数据表示如何产生适当的结果。现在介绍逻辑工作方式。

开始的时候，`main` 谓词将会被直接调用，然后会转向 `run` 谓词，`run` 谓词所要做的第 1 件事情就是读取文件 `fa.txt` 中的数据。一旦数据存在的话，程序就会系统地一个个测试处理数据，并把测试结果写在控制台上。

处理数据的机制是标准的回溯和递归机制。在这里并不详细描述关于 Prolog 如何工作的细节，只是简要给出了其内部工作的原理。

当谓词 `run` 调用谓词 `father` 时，它并不会停在 `father` 谓词结束的地方，在谓词结束时，`fail` 的调用将驱使 Prolog 的推理机在 `father` 谓词中寻找另一个可执行的地方，这种行为就是回溯，因为 Prolog 推理机事实上可以在执行过的程序中回溯查找。

这种情形会循环发生（就像重复或循环的过程），并在每次循环中谓词 `father` 都将产生一个结果，最后在数据中给出的所有可能的 `father` 的定义被耗尽，谓词 `run` 别无选择只能转到谓词 `run` 的下一个子句体。

这个谓词 `father`（和其他谓词一样）被声明为不确定性的，关键字 `nondeterm` 可以被用来声明不确定性的谓词。通过回溯，不确定性谓词可以产生多个结果。

如果在谓词的声明中使用 `no` 这个关键字，谓词就成为仅能给出一个解答的过程模型，在产生第 1 个结果后，`father` 谓词将会停止，再也不能重新进入。而且，在这种不确定性谓词的子句体中必须注意截断（`!`）的用法。如果在 `father` 谓词的子句体的末尾有截断，并重复出现，谓词将只产生一个结果（即使它被声明为不确定性的）。

`Anyflow` 的流模式告诉编译程序，赋予谓词的参量可以是自由的也可以是约束的，没有什么限制。这就意味着和 `father` 谓词的定义一样，有可能同时产生父亲的名字（如果后代的名字是约束的话）和后代的名字（如果父亲的名字是约束的话）。同样可以处理两者都受约束的情况，在这种情况下，谓词会检查这样的关系是否存在于数据中。

明确地说，`anyflow` 流模式同样适用于当所有的 `father` 谓词里的参数是自由变量时的情况。这时，`father` 谓词将在程序学习到的数据中产生多种父子关系的结合体（通过查询 `fa.txt` 文件）。像下面代码显示的那样，最后的用法是在 `run` 谓词中所执行的内容。注意到当 `father` 谓词被调用时，`father` 谓词中的变量 `X` 和 `Y` 并不受约束。

```
run():-
    console::init(),
    stdIO::write("Load data\n"),
    reconsult("fa.txt"),
    stdIO::write("\nfather test\n"),
    father(X,Y),
    stdIO::writef("% is the father of %\n", Y, X),
    fail.
```

7.1.8 小结

Visual Prolog 6 中的程序非常类似于传统 Prolog 中的程序，有许多不同的关键字用来区分一个 Prolog 程序的不同部分。虽然 Visual Prolog 是一种面向对象的语言，但也可以开发不使用面向对象特点的代码，这一节描述了基于控制台的应用程序(`family1`)，并描述了如何构造这样一个程序。

同样发现，通过把部分代码独立于编译的二进制应用程序而保存在数据文件中，可以模拟传统 Prolog 程序的动态工作机制，这种数据文件的句法非常相似于 Prolog 的句法。

7.2 Visual Prolog 的 GUI 编程

这一节将增加一个简单的 GUI（图形用户接口）前端到 `family` 项目，这个项目是在 7.1

节中开发的。这里介绍直接在 Visual Prolog 的 VDE（可视化开发环境）中创建和编辑一个简单 Windows 程序的大多数 GUI 组件、关于 GUI 事件（如单击 GUI 的特定部分上的一个按钮等），以及它们在 Visual Prolog 程序中如何工作等有关说明。另外还将通过两个例子来介绍关于模态对话框的有关知识，一个例子是由 Windows 操作系统内建的，另一个例子是自己构建的程序。

7.2.1 GUI 概述

GUI 是图形用户接口首字母的缩写。在 Windows 操作系统中，这个术语表示人们熟悉的窗口，窗口带有菜单栏、工具栏、窗口右上角的小按钮等。这些元素中的每个元素都称作为一个 GUI 组件，而且在良好设计的程序中，这些组件按照公认规范进行工作。

用程序设计术语来说，一个 GUI 完成两件事。它使用复杂的图形例程在计算机显示器的相应部位上安放和恢复图形图像。例如，一个窗口右上角带有 x 的小方框。它还控制鼠标和其他输入设备在这些图形区域上的行为。幸运的是，这些详细的程序设计是由操作系统完成的。作为一个程序员，并不需要编排这些图形、鼠标和键盘函数，Windows 已经做了这一切。它还提供一个 API（应用程序编程接口），可以用来建立任何程序所要求的 GUI。

此外，Visual Prolog 已经增加了一个更高层的功能——PFC（Prolog 基类），不仅有助于使用 GUI，而且有助于其他领域的编程。

更有甚者，Visual Prolog 的 VDE 可以用来可视化地创建正在开发的程序所用的最终 GUI 实物模型（moch-up）。这里将使用 7.1 节中 family 例子的处理逻辑。

在一个 GUI 程序和控制台程序之间，存在许多差异。然而它们也有类似之处，即控制台程序和 GUI 程序总是从一个固定的入口点（从 goal 或过程）开始。控制台程序不显示图形元素，所以程序由用户提供输入，这些输入或者是来自程序员预先设置的初始参数，或者是来自程序员预先设置的直接询问。程序中各种活动的顺序和方向是由程序员确定的。用户不能够更改该顺序。在一个控制台程序中，程序从 goal 或过程启动，然后逻辑上从那一点开始向前工作，严格地使用户通过由程序员设置的一系列逻辑步骤。

在 GUI 程序中，程序员可以做出灵活的选择。例如，如果启动任何一个常规的 Windows 程序，一般来说它并不强迫人们做这做那。可以随意地浏览菜单而不点击其中可选的任何一项。考虑这样一种特殊的情节：File 菜单有几个菜单项，可以使鼠标光标通过该菜单中的任何一项，而并不完成实际的菜单点选操作。事实上，通过如此这般地在程序的 GUI 上不经意的浏览操作，是人们达到掌控软件的常用方法之一。

无论如何，编写 GUI 程序时，程序员必须使用一种不同的策略。一方面，编程更加容易，因为对所实现的程序中的一切活动不再硬性刻板或严格控制。

但是另一方面，程序员必须知道每个 GUI 组件是如何工作的，什么是常规公认的惯例，以及如何坚持这些习惯，以便当使用这些程序时，用户不会产生不必要的惊奇。同样，有时很难预测由于某些 GUI 事件的触发所产生的结果。因此，程序员必须仔细地分析逻辑，以便无论 GUI 如何使用都能维持逻辑的有效性，仔细地调整这些情形，而程序的逻辑不可以进行调整（使用有礼貌的、易于理解的警告对话框等）。

7.2.2 GUI 对事件的响应

在一个 GUI 程序中，所有的 GUI 组件等待来自键盘或鼠标（或其他输入设备，如数字化仪等）的输入。

来自这种输入设备的信息（鼠标的点击或其他的 GUI 活动）称为一个事件（event）。要求 GUI 程序予以响应的还有其他一些事件——其中一些是由操作系统内部产生的，如内部时钟中断，与当前并发运行的其他程序的交互，ActiveX 事件等。依据程序预期要完成的工作，程序员可以指定适当的代码部分来响应有关的事件。

用标准的计算机术语来说，当程序响应一个事件时，程序通过使用事件处理器已经处理了该事件。因此，一个事件处理器是专门等待和实际处理一个 GUI 事件的一段代码。

事件处理的一个重要特性应当这样理解：

因为一个事件处理器的代码，只是专门等待某一事件的发生，所以看起来似乎毫无逻辑可言。其他模块甚至不必知道在实际处理一个 GUI 事件的类内部秘密地发生的事情。在前面叙述的内容中注意到，包含的所有逻辑都是良好的，它们在逻辑上完整地彼此连接在一起。但是在 GUI 程序的情形，处理 GUI 事件的部分程序代码是被分成片断的，并且被放到了不同的事件处理程序之中。

这一节将采用与前面所举的 family 例子相同的逻辑，经由 GUI 接口使用该逻辑。

7.2.3 开始一个 GUI 项目

现在从零开始一个项目，这是一个非常好的起始点。当在 Visual Prolog 6 的 VDE 中创建一个项目时，确保 UI 策略设置为 GUI，如图 7.9 所示。VDE 则创建最初的处理 GUI 所需要的模块集合和 GUI 组件资源：主菜单、一个顶部的工具栏、一个底部的状态栏、About 对话框和该程序的任务窗口（task window）。

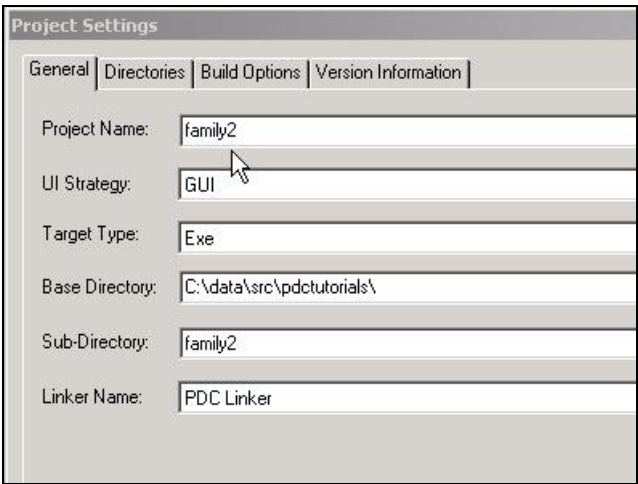


图 7.9 family2 的“项目设置”对话框

创建一个项目之后，就可以立即编译它，这是一个空的 GUI 程序。在这个时期，程序实际上什么也不做。然而，它拥有了一个 GUI 程序所期望的全部功能。Visual Prolog 已经直接构造了一个运行 GUI 程序的基本框架，并且已经提供了一些通常所需要的特性。

例如，在应用程序的主窗口（称作任务窗口 `task window`），Visual Prolog 给出了另一个标题为 `message` 的窗口。这个窗口用来作为内部的控制台程序。当程序员在程序中使用 `stdio::write(...)` 谓词时，结果将直接在 `message` 窗口输出。如果 Visual Prolog 没有将 PFC 类 `stdio` 的输出重新定向到 `message` 窗口，则那些串将不会被看到，因为按照默认约定，一个 GUI 环境没有控制台区。

在一个控制台应用程序中，控制台总是可以作为一个黑板来使用，程序员可以在其上输出信息，这就是为什么这些应用程序被称为控制台应用程序的原因。由前面的章节可以看到，在这样的应用程序中直接使用 `stdio::write(...)` 谓词向控制台输出。

在这一阶段运行所编译的 GUI 程序时，只可以控制程序的菜单、改变程序外部主窗口（即任务窗口）的大小、双击 `message` 窗口在任务窗口之中进行全程缩放等。Visual Prolog 为 `message` 窗口恰好给出一个小的常常是很方便的弹出式菜单。在 `message` 窗口内部任何地方右击鼠标，随即就会弹出一个小菜单，可以清除 `message` 窗口的内容或进行其他的活动。

但是到现在，这个简单的 GUI 程序仍不具备人们所期望的任何逻辑功能，还需要进一步做些工作来获得这种功能。

在着手开发具有程序的实际工作逻辑的项目之前，应该创建或者修改所需要的一些 GUI 组件。在正常环境下，程序员必须花费一些时间针对该程序的 GUI 提出一种策略，构造一个清晰的 GUI 组件的列表，并且只有到了这个时候才着手创建或者修改那些组件。这种活动的完成应该事先进行精心的筹划。但是为了简单起见，在本节中所进行的工作是假定这个规划过程已经完成了的。

在编码阶段，所有 GUI 组件都分别存放在单个的资源文件之中。在其他大多数编程语言中，这些资源文件被分别进行编译，然后在链接过程中把它们包括到主代码里。Visual Prolog 则自动地处理所有这些资源编译和链接问题，而无需用户操心。

7.2.4 创建模态对话框

现在，给程序添加另一个 GUI 组件，供以后需要时使用。这个组件是一个对话框，它用来给程序提供要处理的一个人的名字。在项目树（`project tree`）中，所有模块和资源都清晰地展现在树形菜单里，右击任务窗口，从弹出菜单的上下文菜单中选取 `New` 菜单项，如图 7.10 所示。

创建项目条款的对话框（`Create Project Item`）出现，如图 7.11 所示。确保对话框左面的 `Dialog` 选项被选中，并且在右面输入如图 7.11 所示的详细信息。

这样，一个默认的对话框被创建出来了，如图 7.12 所示，可供进一步编辑加工之用。因为没有为这个对话框实现帮助功能，所以就可以单击一下特定的 GUI 组件（例如，`Help` 按钮），再按下 `Del` 键即可将其删除。

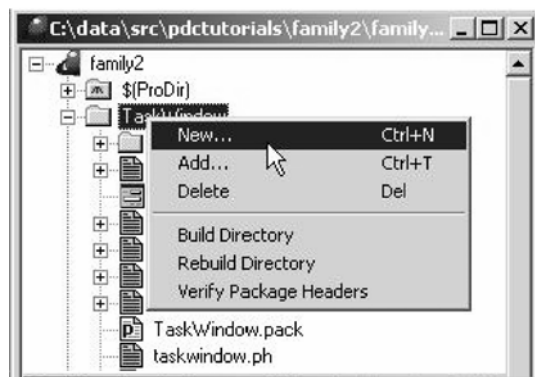


图 7.10 激活“创建项目条款”对话框



图 7.11 “创建项目条款”对话框

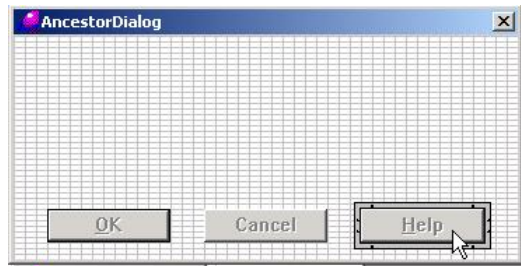


图 7.12 创建的默认对话框

然后，再分别选中其他两个按钮，移动到合适的位置，最后的结果如图 7.13 所示。

通过使用对话框控件，为对话框插入一个静态文本。所谓静态（static）代表一个 GUI 元素是不可点击的。参考前面提到的图像，在单击静态文本创建按钮之后，可以在对话框中画一个矩形区域，所要输入的文本文字将出现在其中，如图 7.14 所示。

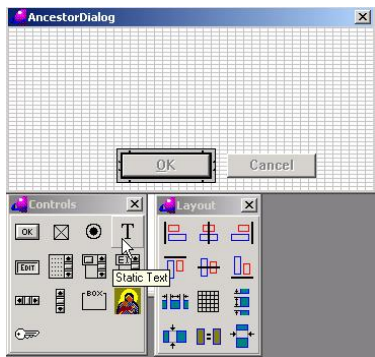


图 7.13 编辑默认对话框

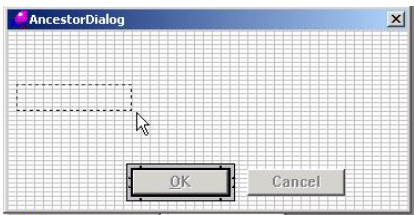


图 7.14 在默认对话框中增加控件

随之编辑控件属性（edit control attributes）将出现，如图 7.15 所示。在 Text 字段下，输入 Person，所输入的文本将出现在对话框里面。

以类似的方式，使用对话框编辑控件来插入一个可编辑的文本字段或一个编辑控件，如图 7.16 所示。

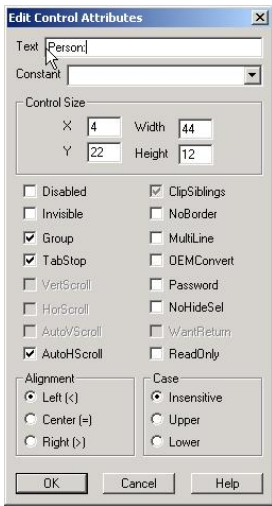


图 7.15 “编辑控件属性”对话框



图 7.16 “编辑控件”工具栏

此时，先放下那个空的文本字段，转去改变常量（constant）符号为某个有意义的符号常量，而不是由 VDE 提供的默认常量，这个步骤如图 7.17 所示。这个常量在程序代码内部使用，以便程序代码引用这个编辑字段。

构造的对话框的最终情形如图 7.18 所示。

每个对话框有一个属性称为访问次序。需要完成的最后一步，就是设置这个控件在对话框中被访问的次序，即当用户按下 Tab 键时，对话框中的 GUI 组件与用户交互的次序。短语“访问控件（visit a control）”意思是说控件获得输入焦点，这是一个十分专业的术语。短语“控件接收输入焦点（the control receiving the input focus）”意思是说控件立即从键盘接收输入。例如，如果一个编辑字段具有输入焦点，就可以看到该编辑字段中闪烁的插字符号，表示该字段准备接收经由键盘输入的字符。

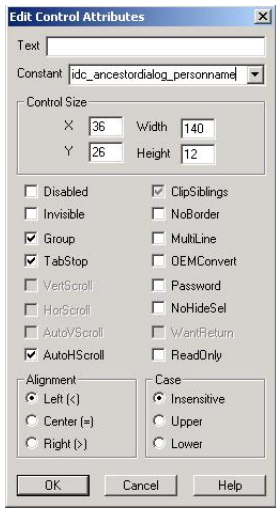


图 7.17 在“编辑控件属性”对话框中改变常量

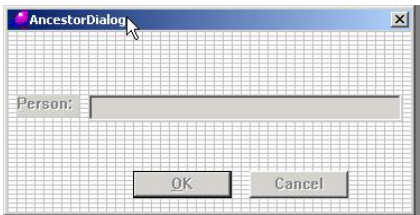


图 7.18 创建的对话框

为了改变访问次序，右击该对话框，在随即弹出的菜单中选取“访问次序 (visit order)”菜单项。

一些小按钮将出现在该对话框的 GUI 组件上，如图 7.19 所示。在这里可以看到，文本编辑控件 (idc_ancestordialog_personname) 的访问次序为 3，而 OK 按钮的访问次序编号为 1。

这就是说，当该对话框呈现给用户的时候，该文本编辑控件将不会立即接收键盘的字符。当用户按一下 Tab 键时，焦点将转移到 Cancel 按钮，只有再次按下 Tab 键时，焦点才转移到该文本编辑控件。只有到了这个时候，该文本编辑控件才接收键盘的输入。

总之，相对于该对话框中的其他控件而言，编辑字段的访问次序编号是最后一个。

当然也可以改变这种情形。单击标识为 3 的小按钮，这将打开“访问次序 (Visit Order)”对话框，如图 7.19 所示。可以用“+”和“-”来改变访问次序编号。在这个例子中，将改变这个访问次序编号，使得该文本编辑控件 (idc_ancestordialog_personname) 的访问次序为 1。这样，当这个对话框在程序中被使用的时候，输入焦点将首先落在该文本编辑控件之上。

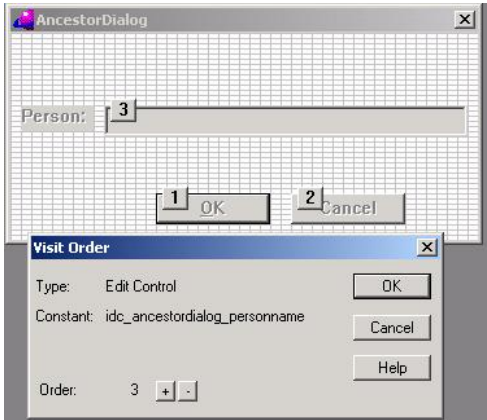


图 7.19 确定对话框上控件的访问次序

注意：在对话框中，有另外一个属性称为默认控件（default control），它可以和访问次序分开来进行设置。一旦按下 Enter 键，已经被设置为默认控件的控件事件将会被触发，而不管当前哪个控件具有输入焦点。通常，默认控件设置为 OK 按钮。

当右击对话框时，可以通过使用如图 7.20 所示的对话框设定其属性。值得注意的是，对话框的类型 Type 项被设置为模态（modal）。模态是指只要对话框呈现给用户，GUI 就会停止对程序其他部分的响应。只有当用户关闭这个对话框（通过单击 OK 或 Cancel 按钮），GUI 才会再次响应所有的 GUI 活动。

相反，非模态（modeless）对话框没有这个限制。即使对话框处于激活状态，整个 GUI 界面也是响应的。创建一个非模态对话框需要一些更多的考虑，在此不做进一步介绍。



图 7.20 对话框属性

使用同样的“对话框属性（dialog attributes）”（参见图 7.20），也可以把标题改为“Ancestor of ...”。

7.2.5 修改菜单

现在修改程序的主菜单。如前所述，VDE 已经提供了一种在许多程序中都可见到的由标准菜单项组成的默认菜单。要去编辑它以满足程序的功能需求。通过项目树，双击 TaskMenu.mnu 项，可以看到如图 7.21 所示的情形。

这将打开菜单编辑器。菜单编辑器的主对话框如图 7.22 所示。在主目录中选定“编辑（Edit）”菜单后，单击“属性（attributes）”按钮。

接着打开了“菜单项属性（menu item attributes）”对话框，如图 7.23 所示。可以从 &Edit 到 &Query 改变名字，Q 前面的“&”标记指在主菜单中该字母下面有下划线，用户可以通过该字母迅速进入菜单。

可以单击上述对话框中的 Test 按钮来测试这个特性，VDE 的顶端菜单将会暂时被现在设计的菜单所替换，然后可以浏览，感觉其效果。按 Esc 键就可以返回 VDE 菜单的原来

状态。

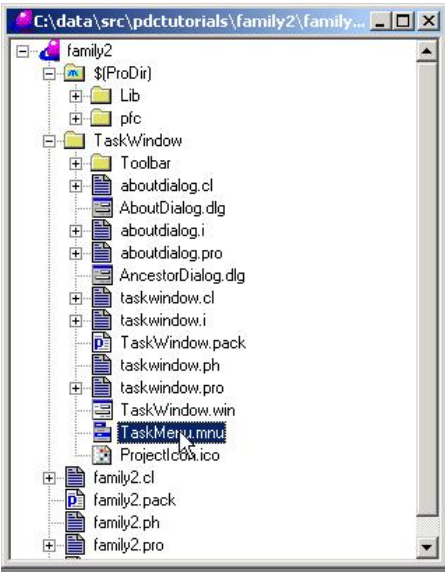


图 7.21 双击任务菜单项

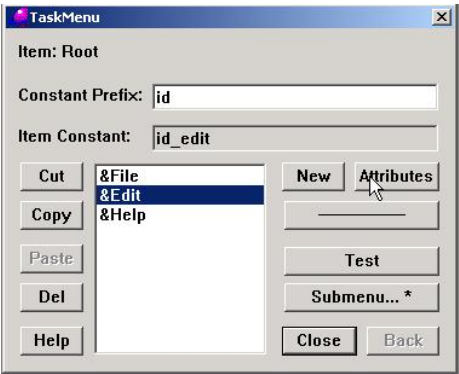


图 7.22 菜单编辑器

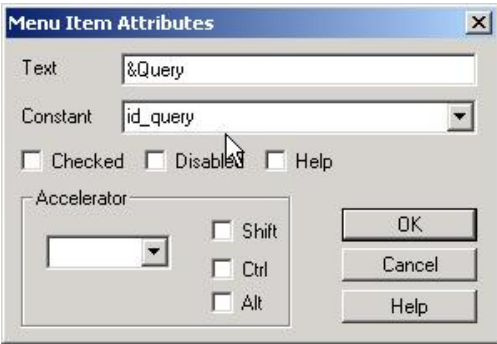


图 7.23 “菜单项属性”对话框

现在返回到“主任务菜单（taskmenu）”对话框。双击 **&Query** 入口，注意到它仍然包含老的 Edit 菜单里的菜单项(Undo, Redo, Cut, Copy 与 Paste)。从主菜单条目入口 **&Query** 删除在里面看到的所有菜单项。接着把常量前缀 **Constant Prefix** 设定为 **id_query**。常量前缀 **Constant Prefix** 由 VDE 内部使用，用于描述表示各种菜单项的常量。这些常数被用来在代码内引用菜单项，如图 7.24 所示。

现在要在 **Query** 主菜单下增加一些菜单条目。单击 **New** 按钮，然后输入信息，进入如图 7.25 所示的界面。

注意到常量 **Constant** 的值依赖于为菜单项输入的文本而被自动创建。在上述例子中，常量值是 **id_query_father**，就是在上面图形中所看到的那个常量。同样可以创建 **&Grandfather** 和 **&Ancestor of ...**的菜单项目。

注意，按照约定，省略号（3 个点）放在菜单项的末尾，表示该菜单项在执行某一工

作前要输出一对话框。在上面的例子中，&Father 和&Grandfather 菜单栏没有省略号。但是 &Ancestor of ...菜单项有省略号，因为一旦调用这个菜单项，在其他行为执行前会打开一个对话框。

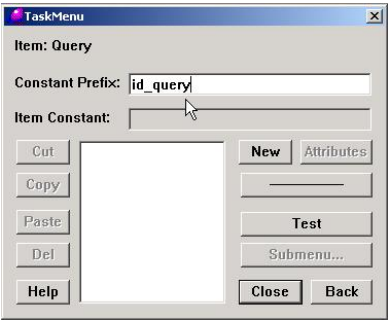


图 7.24 在“任务菜单”对话框设定常量前缀

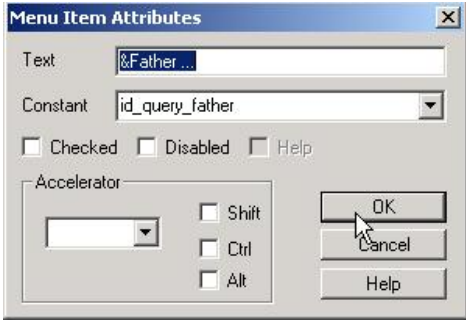


图 7.25 利用菜单项属性增加子菜单

最后一步是 TaskWindow 菜单的编辑。在默认状态，当 VDE 创建菜单时，File→Open 菜单无法使用，必须找出该菜单项，从 Menu Item Attributes 对话框中去掉禁止标记，使它变为可用，如图 7.26 所示。

顺便提一下，在前面的例子中可能注意到，可以设置用户调用的加速键（热键等），来快速调用与菜单项一样的功能。在上面的例子中，F8 就是功能键。应该注意，在这里打开文件的菜单项为&Open（没有省略号）。应当纠正它为&Open...以适应省略符号的约定。

当关闭 TaskMenu 对话框时，VDE 会请求确认是否想保存这个菜单。单击 Save 按钮保存。

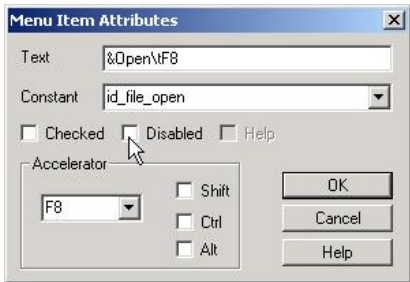


图 7.26 “菜单项属性”对话框

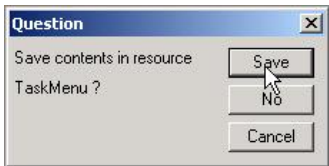


图 7.27 VDE 确认对话框

7.2.6 修改工具栏

工具栏的应用是 GUI 的另一个有用部分。通常，它包括具有各种菜单功能的按钮，简单地说，这些按钮就是这些菜单的快捷方式。现在编写工具栏的程序。当开始创建一个项目时，Visual Prolog VDE 会为程序创建一个默认的工具栏。在项目树中双击 ProjectToolbar.tb，如图 7.28 所示。

这将调用工具栏编辑器，如图 7.29 所示。

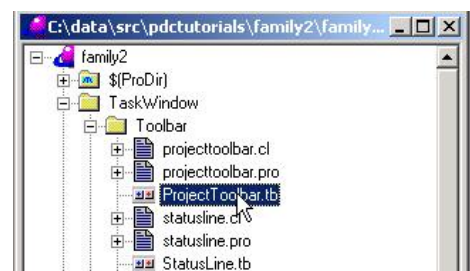


图 7.28 在项目树中双击工具栏项



图 7.29 “项目工具栏”编辑器

在图 7.29 中，顶端是正在编辑的工具栏，底端是编辑工具栏组件可以用到的各种控制按钮。

在工具栏中，包含一组有预定图标图形的按钮（与一般 GUI 程序一样）。如果希望，也可以改变这些图标的图形。需要指出的是，Visual Prolog 的 VDE 内部已经恰当地包含了一个非常好的图标编辑程序，对大的图标，VDE 打开 MS 画图工具来编辑。

这些按钮已经被定制为一套功能菜单项。但是当编辑菜单项时，也要编辑工具栏按钮，并且把它放在适当的位置。

首先，对剪切、复制和粘贴等按钮进行删除处理，因为程序没有这些功能。为此，首先选择这些按钮，然后从工具栏中删除。删除以后，工具栏如图 7.30 所示。

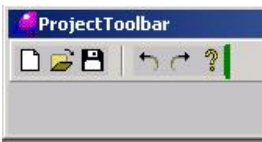


图 7.30 项目工具栏按钮

现在把 Undo, Redo 和 Help 按钮映射，并表示为下列菜单项 Query|Father..., Query|Grandfather...和 Query|Ancestor of ...。

前面提到过，在本章中将不改变那些按钮的图标图像。

双击 Undo 工具栏按钮，在出现的“按钮属性 (Button Attributes)”对话框内，将象征工具栏按钮的内部常量从 id_edit_undo 变为 id_query_father，如图 7.31 所示。

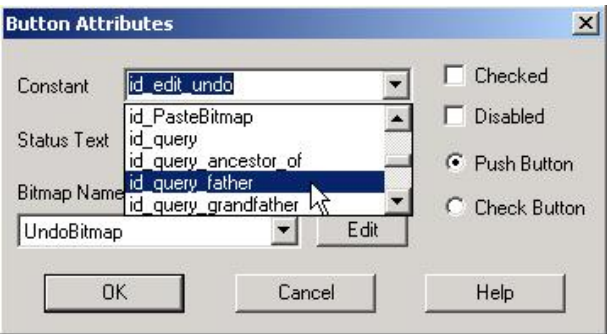


图 7.31 “按钮属性”对话框

在同一个对话框中，还应该将 Status Text 设置由

Undo ; Undo

变为

```
Query fathers;List of all fathers listed in the database
```

上一行的分号将字符串分成两部分，第 1 部分作为按钮本身的工具提示而显示，第 2 部分将在主窗口的状态行显示。

运用同样的方法，改变 Redo 按钮的常量值为 id_query_grandfather，改变 Help 按钮的常量值为 id_query_ancestor_of。这两个按钮的状态行内容也做适当的修改。

7.2.7 在程序中添加主代码

已经完成了程序中需要的全部 GUI 功能性工作。现在开始插入逻辑代码。在做这些工作之前，先来了解常用的程序运行方式和现在的 GUI 程序运行方式之间的一些重要差异。

当一个用户使用一个 GUI 程序时，就像用户使用一个房间。用户正好通过大门进入房间，但是一旦进入房间，用户就可以随便决定房间的哪一部分要被使用。毋庸置疑，程序员对程序部件的每一部分如何工作最具有发言权，但是程序的哪一部分被激活大体上由用户自由决定。目标程序仍然在 GUI 程序中出现，但是，在这里它所做的一切只是为用户的工作搭建一个平台，就像进入房间的主门一样。

这对程序员来说意味着什么呢？模块的内部已经包含了初级逻辑代码。但是，现在依靠相应的 GUI 组件控制的逻辑，它将被扩展到几个模块。

先来增加一些程序必需的逻辑代码。打开 TaskWindow.pro，如图 7.32 所示，将插字编辑符定位在下面的代码行：

```
facts
    thisWin : vpiDomains::windowHandle := erroneous.
```



图 7.32 任务窗口代码

一旦找到那一行代码且设置了插字编辑符（参见图 7.32），则在此插入下列代码：

```
domains
    gender = female(); male().

class facts - familyDB
    person : (string Name, gender Gender).
    parent : (string Person, string Parent).

class predicates
    father : (string Person, string Father) nondeterm anyflow.
clauses
    father(Person, Father) :-
        parent(Person, Father),
        person(Father, male()).

class predicates
    grandfather : (string Person, string Grandfather) nondeterm anyflow.
clauses
    grandfather(Person, Grandfather) :-
        parent(Person, Parent),
        father(Parent, Grandfather).

class predicates
    ancestor : (string Person, string Ancestor) nondeterm anyflow.
clauses
    ancestor(Person, Ancestor) :-
        parent(Person, Ancestor).
    ancestor(Person, Ancestor) :-
        parent(Person, P1),
        ancestor(P1, Ancestor).

class predicates
    reconsult : (string FileName).
clauses
    reconsult(Filename) :-
        retractAll(_, familyDB),
        file::consult(Filename, familyDB).
```

上面的代码是程序的核心逻辑部分。在本节，直接将它们插入 `TaskWindow.pro` 模块中，因为要了解 GUI 组件如何进入到程序的核心逻辑中执行各种动作。在更复杂的例子中，核心逻辑代码会分开存放，有时分散到好几个模块中。GUI 模块（如 `taskwindow`）通过扩大模块的范围，分别引用这些模块。

事实上，让程序的核心逻辑代码尽可能地分散到所有 GUI 相关的模块中是一个很好的

实践，但在本节中将有意忽略这些。

7.2.8 压缩相关代码

既然已经考虑了核心逻辑，现在需要插入交互位（interactive bits）。在项目树中右击 TaskWindow.win 条目，该条目表示该主任务窗口的窗口资源。所有发生在这个窗口的单击、菜单等事件都由写给这个窗口的事件处理器处理。右击所选的项，弹出如图 7.33 所示的上下文菜单，从该菜单选择“代码专家”菜单项。

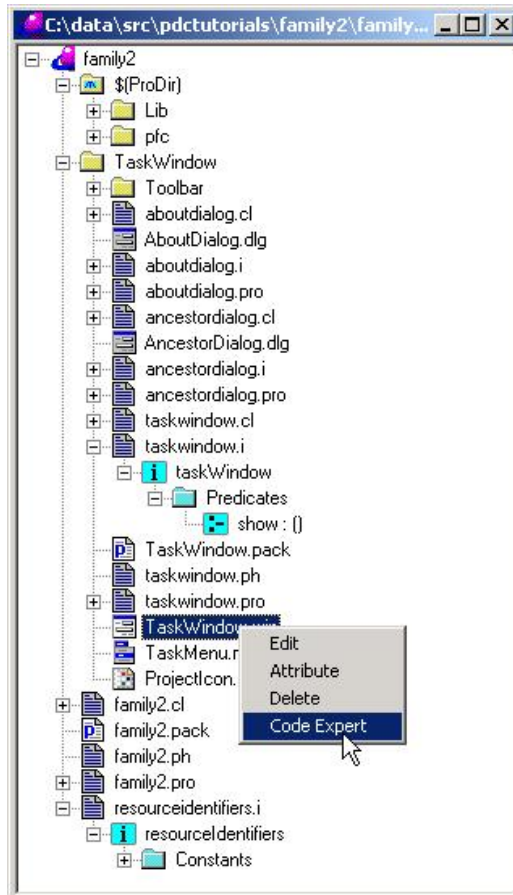


图 7.33 在右键弹出菜单中选择

对话框和窗口专家（dialog and window expert）如图 7.34 所示，它为特定窗口或对话框控件（交互式 GUI 组件）设置默认的事件处理器。蓝色填充的圆圈表示没有给定控件的事件处理器，绿色的勾记号表示存在相关的事件处理器。

使用上述对话框，确保事件处理器是为常量 `id_file_open` 所表示的菜单项设置的。

现在，在项目树中单击 TaskWindow.pro 模块，用 Build 菜单来编译它。如果 TaskWindow.pro 模块没有被选择，那么 Build 菜单中的 Compile 菜单将不可用，因此一定要谨慎。

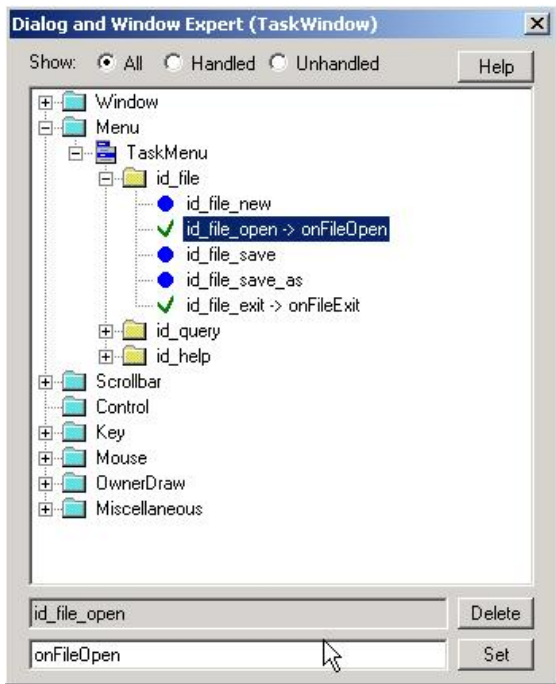


图 7.34 对话框与窗口专家

当用 Visual Prolog 6 编译模块时，它要改造项目树，使所有相关的谓词和论域等清晰地展开，便于直接访问。因此要使用这些特点来保证 VDE 列出在 TaskWindow.pro 模块中定义的所有谓词。然后即可对谓词进行操作。

注意：VDE 自动识别出 TaskWindow.pro 需要一些附加模块，如图 7.35 所示，对该模块的编译进行了两遍。

编译完成以后，当引用 TaskWindow 谓词时，会在项目树中看到所有的谓词，如图 7.36 所示。

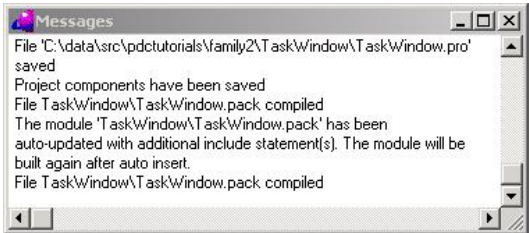


图 7.35 消息窗口

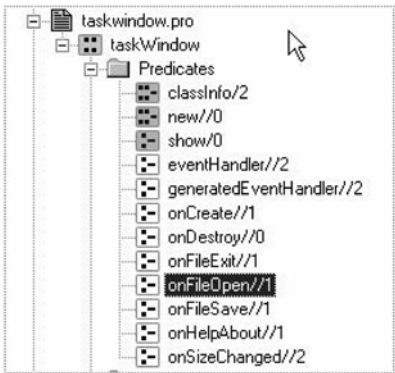


图 7.36 项目树中的谓词

可以发现 taskwindow.pro 模块的谓词部分将显示出谓词 onFileOpen，而原来它是不存

在的。

双击该谓词，就会直接打开编辑器，并定位于谓词 `onFileOpen` 的子句位置处。当该谓词有多重子句时，指针将指向第 1 个子句体。

谓词 `onFileOpen` 被称作事件处理器（windows 应用程序术语），作为程序员，不需要访问这个谓词。对应的 GUI 组件被激活时，windows 会自动调用它。在这里单击该菜单项，默认情况下为事件处理器插入如下代码：

```
onFileOpen(_MenuTag) = handled(0).
```

用下面的代码替换该代码如下：

```
onFileOpen(_MenuTag) = handled(0):-
    Filename = vpiCommonDialogs::getFileName(
        "*.txt", ["Family data files (*.txt)","*.txt",
        "All files", "*..*"],
        "Load family database",
        [], ".", _),
    ! ,
    reconsult(Filename),
    stdIO::writef("Database % loaded\n", Filename).
onFileOpen(_MenuTag) = handled(0).
```

如果检查上面的代码，就会发现，只不过在 VDE 给定的一个子句体上又增加了一个子句体。第 1 个子句体将从加载数据库的地方打开标准的 Windows 对话框。

第 2 个子句体提供一个失效——安全机制，当取消对话框时它就会执行。它告诉程序 GUI 事件已经安全处理。如果这条语句没有处理，该事件就会回溯到其上一层窗口，由上一层窗口的 GUI 组件处理，这种 GUI 事件由一个 GUI 组件到另一个高层次 GUI 组件的自动传播是 GUI 环境（比如 Windows）的特有机制。如果不知道某个事件在哪里处理，最好让 Visual Prolog 决定事件的默认处理。

如果现在编译并运行它，就可以使用 `File|Open` 菜单项把 `family` 数据库装载到程序中。为了测试它，可以使用为这个程序的控制台版本开发的 `fa.txt` 数据库，这时，消息窗口的信息会发出一个通知，指示程序装载数据库是否成功。

注意：尽管这个对话框会要求有 `.txt` 文件，但是不应该把它们和计算机中使用的普通文本文件相混淆。这些文本文件会自动被格式化为程序所要求的形式，其他类型的文件会导致错误。

如果文件已经被正确加载，那么要调用 `stdIO::writef(...)` 谓词输出结果。一个 GUI 程序没有正常的控制台，但是 Visual Prolog GUI 程序会自动提供一个消息窗口作为输出控制台，因此 `stdIO::writef(...)` 谓词结果会在消息窗口中出现。如果已经关闭了该窗口，那么就看不到输出的结果。可以根据自己的爱好来调整该窗口的大小。

对于对话框和窗口专家（Dialog and Window Expert），要确保 `Query | Father`，`Query | Grandfather` 和 `Query | Ancestor of ...` 菜单项的事件处理程序设置如图 7.37 所示。

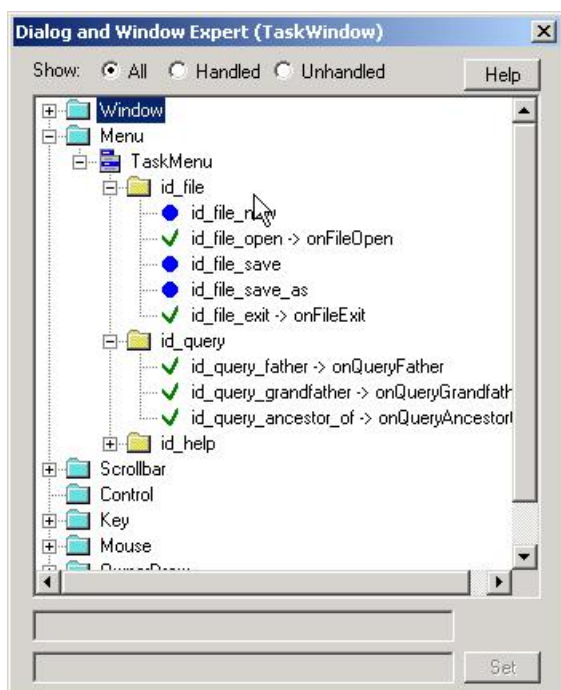


图 7.37 设置菜单项的事件处理程序

对于 Query|Father 菜单项，在 Visual Prolog 自动生成的子句体前添加如下子句体：

```
onQueryFather(_MenuTag) = handled(0):-
    stdIO::write("\nfather test\n"),
    father(X, Y),
    stdIO::writef("% is the father of %\n", Y, X),
    fail.
```

对于 Query|GrandFather 菜单项，在 Visual Prolog 自动生成的子句体前添加如下代码：

```
onQueryGrandFather(_MenuTag) = handled(0):-
    stdIO::write("\ngrandFather test\n"),
    grandfather(X, Y),
    stdIO::writef("% is the grandfather of %\n", Y, X),
    fail.
```

7.2.9 分析所做的工作

那么哪里是封装呢？其实这里已将代码分成两部分，前面所讲到的是非交互式的逻辑核心。而这部分需要接收用户的输入，并且这些部分被存储为不同的事件处理器，至于其他的程序，它并不需要知道这些事件处理器在程序中怎样编写，下面添加一些交互代码到其他事件处理器。

在 Visual Prolog 自动产生的默认子句体前，为 Query | Ancestor of... 菜单项增加如下子句体：

```
onQueryAncestorOf(_MenuTag) = handled(0):-
    X = ancestorDialog::getName(thisWin),
    stdIO::writef("\nancestor of % test\n", X),
    ancestor(X, Y),
    stdIO::writef("% is the ancestor of %\n", Y, X),
    fail.
```

上面代码的目的就是从前面已经构造的 `ancestorDialog` 所提供的模态对话框中获取一个字符串。代码的谓词部分假设有一个全局可达的叫做 `getName` 的谓词，该谓词在 `ancestorDialog` 模块中可用，它将会返回某人的名字，这个人的祖先正是要寻找的对象。

与在 `onFileOpen` 事件处理器中看到的一样，寻找一个从模态对话框返回的字符串（文件名）。模态对话框本身被谓词 `vpiCommonDialogs::getFileName(...)` 调用。它与从 `ancestorDialog` 获得字符串的过程采用了相同的策略。惟一不同的是，`vpiCommonDialogs::getFileName(...)` 提供了一个内置的标准的窗口模式对话框文件，但是对于自己家族的 `ancestorDialog`，则不得不做更多的编码工作。

编译这个程序时，可能会出现一个错误，如图 7.38 所示。

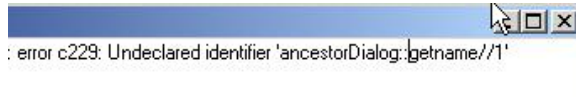


图 7.38 编译错误信息

错误的原因是因为谓词 `onQueryAncestorOf` 所期盼的全局可达谓词 `getName` 从 `ancestorDialog.pro` 模块进行调用。但是还没有那样写，下面来纠正这些错误。

这个谓词在一个模块中定义，但是在其他模块中调用，因此需要确保这种声明不保存在 `.pro` 文件中，但是放在一个所有程序都能调用的位置。模块的类声明文件（扩展名为 `cl`）就是这样的位置。所以，打开 `ancestorDialog.cl`，添加下面的代码（这个声明表示 `getName` 谓词在模块内执行，也能被其他模块调用）。

```
predicates
    getName : (vpiDomains::windowHandle Parent) -> string Name determ.
```

在模块 `ancestorDialog.pro` 里，添加该模块的相关核心逻辑。与 `taskwindow.pro` 设计的方式一样，就是将这些代码插入下面的代码行前：

```
facts
    thisWin : vpiDomains::windowHandle := erroneous.
```

下面就是要插入的代码：

```
domains
    optionalString = none(); one(string Value).
```

```

class facts
    name : optionalString := none().
clauses
    getName(Parent) = Name :-
        name := none(),
        Dlg = ancestorDialog::new(),
        Dlg:show(Parent),
        one(Name) = name.

```

现在还有最后一个问题。需要改变 OK 按钮的事件处理器，这是必须的，以便对话框把由用户输入的值声明到 **name** 事实类。不做这些，上面的谓词将会发现一个空的字符串。

Visual Prolog 给出的默认代码如下：

```

predicates
    onControlOK : vpiDomains::controlHandler.
Clauses
    onControlOK(_Ctrl, _CtrlType, _CtrlWin, _CtrlInfo) = handled(0):-
        vpi::winDestroy(thisWin).

```

上面的代码将被改写如下：

```

predicates
    onControlOK : vpiDomains::controlHandler.
clauses
    onControlOK(_Ctrl, _CtrlType, _CtrlWin, _CtrlInfo) = handled(0):-
        EditCtrl = vpi::winGetCtlHandle(thisWin,
            idc_ancestordialog_personname),
        Name = vpi::winGetText(EditCtrl),
        name := one(Name),
        vpi::winDestroy(thisWin).

```

现在已经完成了程序。如果编译并运行这个程序，将不会有任何错误。

7.2.10 运行程序

一旦开始执行程序，要注意到不会立即执行这些活动。前面已经介绍了控制台程序的工作方式。像前面说的一样，开始一个 GUI 程序就像要进入一个房间，在那里用户可以自由地完成选择的任何事情。每个房间仅仅是等待用户决定给出的输入。

类似地，在这个小程序中，可以不装载任何数据调用 Query|Query Father 菜单。它不会产生任何结果或错误，因为主要逻辑只关心默认数据的情况，可以在选项中调用 File|Open... 菜单，并装入 family 数据库（前面的教程中用了同样的一部分）。

然后，可以测试 Query | Query Father, Query|Query Ancestor 和 Query | Ancestor of... 来查看控制台程序获得的上述结果。这些结果将被显示在消息窗口（注意，不要关闭消息

窗口，否则结果不会被显示出来）。当发现可以多趟运行询问时，GUI 程序将被认可，那么可在任何时间点上自由地调用不同数据。

7.2.11 小结

在这一节，研究了 GUI 的基本元素以及如何用 Visual Prolog 开发一个 GUI 程序。可以发现 Visual Prolog 给出的 VDE 允许完全控制希望使用的所有 GUI 组件。可以编辑菜单、对话框和工具栏，以满足所希望的功能。

可以模块化 Prolog 代码，并且将代码插入程序的不同部分，以便它们被安全地封装到不同的事件处理器中。非交互式逻辑核心被分别保存，因此开发的 GUI 程序可以被灵活使用，也不强迫用户关心程序执行活动的顺序。

7.3 Visual Prolog 的逻辑层

在这一节，将修改在7.2节开发的简单GUI（图形用户接口）程序，并将程序的逻辑从其余的GUI代码中隔离出来。这一代码逻辑会被放入一个属于自己的程序包的对象内。而隔离代码的方法会使人们清楚地从GUI中去思考（代码）逻辑问题。

7.3.1 初始准备阶段

首先需要以正确的步骤安装项目的GUI，假定读者已经正确完成了那些GUI构造过程，因此这里不再重复。

注意：假定读者已经在GUI中创建了AncestorDialog对话框，并且已给定了代码，因此该模块已经定义了谓词getName来检索在该对话框输入的祖先的名字。

假设该项目名字为family3。

程序的GUI在技术上称为表示层。较早的时候，逻辑代码被混合在表示层内。现在，将取出该逻辑并把它放入项目内自己的程序包中。此程序包可用作业务逻辑层（business logic layer）。

注意：是使用句子business logic而不是logic，因为它指示程序的business层面。即程序存在的理由来自于逻辑层。

因为Visual Prolog 6是一种强有力的面向对象语言，业务逻辑会被表示为一个对象。关于深入地理解Visual Prolog 6面向对象编程的细节，读者可参见“类与对象”一章中的详细解释。但为了完整起见，有关面向对象的一些思想在本节中也会提到。下面逐步完成程序。

7.3.2 创建业务逻辑层

一旦安装好family3项目的GUI，就会看到如图7.39所示的主项目目录。

右击目录的顶端，从弹出的菜单中选择New菜单项，如图7.40所示。

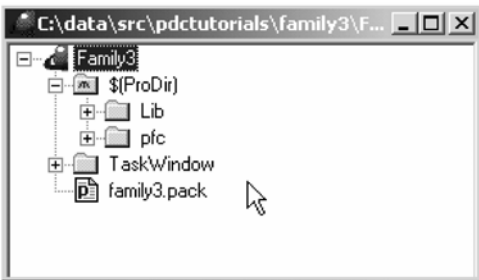


图 7.39 family3 的项目树

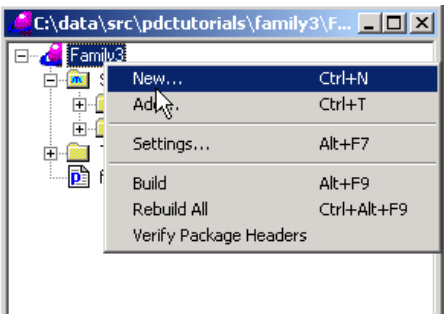


图 7.40 在 family3 项目树中的右键菜单

在出现的“创建项目条款 (create project item)”对话框中，如图7.41所示，选择Package（左边列表的第1项），并且给定package的命名为familyBLL（为使家庭业务逻辑层容易记忆，应该选择有意义的名字）。

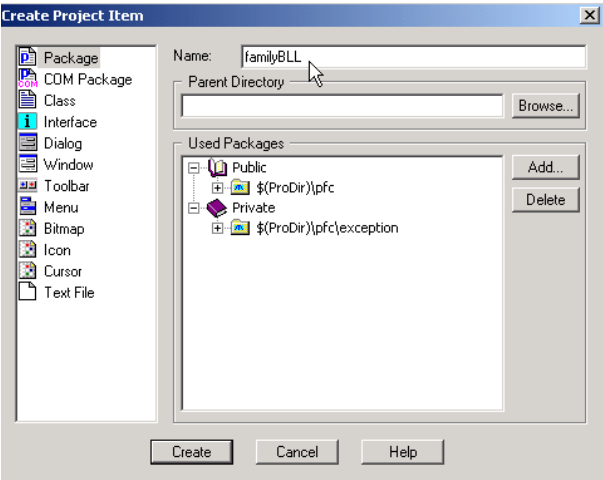


图 7.41 在“创建项目条款”对话框键入程序包的名字

单击Create按钮，项目目录呈现出的内容如图7.42所示。

如果现在查看硬盘上存储这些文件的文件夹，那么就会看到，VDE已经在主项目文件夹内创建了一个单独的文件夹（叫做familyBLL）存储某一程序包的所有文件。

现在建立程序，以确保工作有序完成。

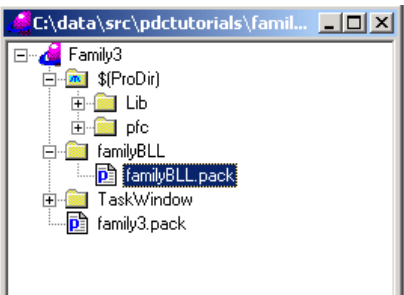


图 7.42 在 family3 项目树中的新内容

7.3.3 在业务逻辑层上工作

在适当的位置放置逻辑。充分利用 Visual Prolog 6 面向对象的特性产生且设计

familyBLL程序包的一个familyBL类。在某些叙述中，有些特征并没有说明。然而这里将使用面向对象系统的主要特征之一，即数据封装。

数据封装指将数据隐藏在某一特定模块的代码内，程序的其他模块不能直接读取或修改其中的数据。数据存取被严格控制，并且数据只能间接地通过该类的谓词读取或修改。

这一原理避免了许多程序设计语言中最普遍犯的程序设计错误。当数据被保存下来为项目的所有程序设计者所读写时，程序设计时常无法快速写出访问代码和改变此类的数据。这是非常成问题的，尤其是在非同步的程序中，比如GUI程序，数据在哪里改变无法预知，时常依照人的行为使用程序。

数据封装的另外一个优点是，内在的数据表达系统能随时在类内被修正，项目内其余的代码不会受到影响。

7.3.4 创建业务逻辑类

在项目目录中的familyBLL程序包文件夹上右击。从打开的菜单中选择New菜单项，会看到如图7.43所示的情形。

在出现的“创建项目条款”对话框中，创建一个名为 familyBL 的类，确保 Creates Object 复选框选中，可以看到如图 7.44 所示的情形。

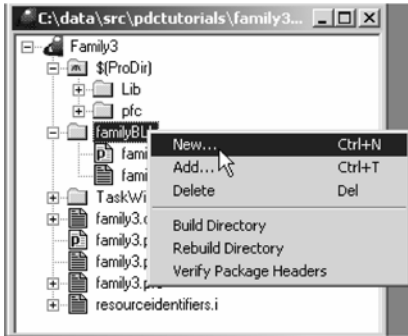


图 7.43 在 family3 项目树中的新内容

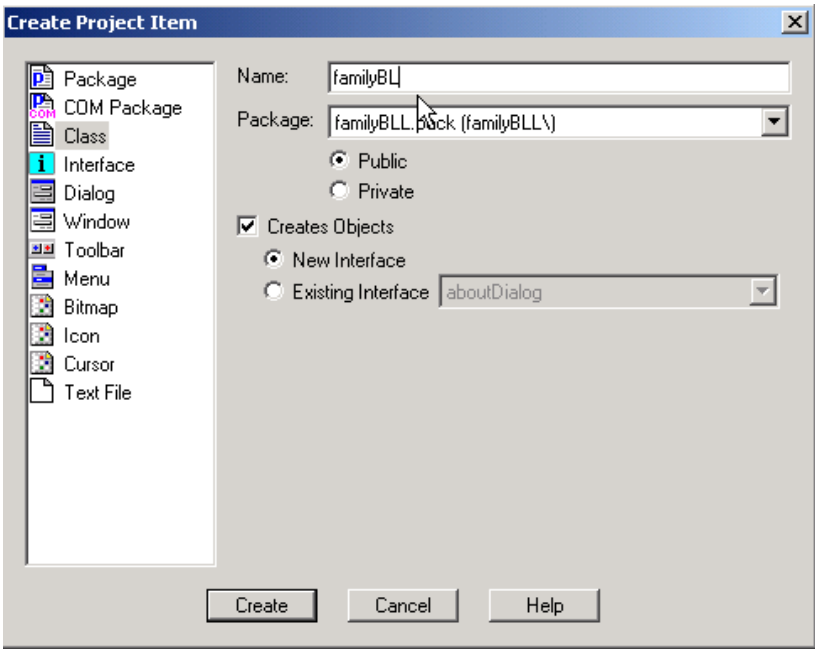


图 7.44 为项目创建 familyBL 类

单击 Create 按钮, VDE 给出 familyBL.pro 文件让编辑。替换从 **implement** 到 **end implement** 间的默认代码为下列代码:

```
implement familyBL

    open core

    constants
        className = "FamilyBL/familyBL".
        classVersion = "$JustDate: $$Revision: $".

    clauses
        classInfo(className, classVersion).

    facts
        fileName : string.
    clauses
        new(Filename):-
            filename := Filename,
            file::consult(Filename, familyDB).

    clauses
        save():-
            file::save(filename, familyDB).
    facts - familyDB
        person : (string Name, gender Gender).
        parent : (string Person, string Parent).

    clauses
        father(Person, Father) :-
            parent(Person, Father),
            person(Father, male()).

    clauses
        grandfather(Person, Grandfather):-
            parent(Person, Parent),
            father(Parent, Grandfather).

    clauses
        ancestor(Person, Ancestor):-
            parent(Person, Ancestor).
        ancestor(Person, Ancestor):-
            parent(Person, P1),
            ancestor(P1, Ancestor).
```



```
end implement
```

现在，选择VDE的File→Open，打开文件familyBL.i，用下面的代码替换默认代码：

```
interface familyBL

open core

domains
    gender = female(); male().

predicates
    father : (string Person, string Father) nondeterm (o,o) (i,o).

predicates
    grandFather : (string Person, string Grandfather) nondeterm (o,o) .

predicates
    ancestor : (string Person, string Ancestor) nondeterm (i,o).

predicates
    save: ().

end interface
```

由VDE自动生成的很多文件需要被修正。选择VDE的File→Open，并且打开familyBL.cl文件。

在文件中见到的谓词段之前，需要插入以下代码：

```
constructors
    new : (string Filename).
```

在插入上述代码之后，文件会是下面这样：

```
class familyBL : familyBL
    open core
    constructors
        new : (string Filename).
    predicates
        classInfo : core ::classInfo.
        % @shot Class information predicate.
        % @detail This predicate represents inform
        % @end
end class familyBL
```

现在，建立项目并检查是否一切正常，此时即使代码在适当的位置，当还没有插入胶

合代码以连接GUI到事务逻辑层的时候，项目会从技术上被编译进一个空的项目之内。但在那之前，先要理解为什么将familyBL类放入3个不同的文件。

7.3.5 理解业务逻辑类

如果比较现在编写的代码与早期教程中的代码，可能会得到如下结论：

- (1) 代码被分成 3 个段。
- (2) 一些谓词和论域被声明在*.i* 文件中。
- (3) 所有的实现被放在*.pro* 文件中。
- (4) *.cl* 文件包含特殊谓词 **constructor** 的声明。

当要求VDE建立一个类的时候，通常指出类会产生对象(objects)。区别一个类和一个对象之间的不同点是很重要的。

打个比喻，一个类可以被看成一个生产并封装比萨饼的自助餐厅。仅仅有一个餐厅不能保证那里就能给出比萨饼，需要有明确的指示让餐厅送出比萨饼。

正如自助餐厅有生产比萨饼的能力，一个类有产生对象的能力。一个对象包含代码和数据，这些代码和数据按照类中给出的实现放在一起。因此类就类似于自助餐厅，它能产生对象。但还不仅仅是这样。

尽管类是在项目中被编码，但它并不意味着项目自动包含类的对象。在每个被设计的用于产生对象的类中，那里肯定会有一个或多个特殊的构造函数**constructor**，它将构建一个包含所有相关代码和数据的预先封装好的对象。

Visual Prolog也允许创建不生成对象的类，在这些类里面的代码主要为模块化目的服务。这一章不讨论不生成对象的类。

类的构造函数写在*.cl*文件中。可以自由存取整个类的谓词（与类产生的对象相反）也存在于该文件中。这可以在**classInfo**实用谓词中看到，它也是在该文件中声明。

创建的每个对象需要有一些可公共存取的谓词操纵封装在它里面的数据。这样的谓词被分别定义在相关类的*.i*文件中。

最后，执行文件（*.pro* 扩展）将包含实际代码。对于所有的实际用途，代码在其他模块中是不可见的。

所有这些工作看起来费力，而事实上正好相反。清晰简洁的处理方式在实际中很容易使用。以这种方式，面向对象程序设计已经在Visual Prolog 6中得到实现。举例来说，分配不同的程序包给不同的程序员通常是一个好的策略，这些程序员将利用面向对象的方法学，以最少数量的障碍将复杂的项目缝合起来。

现在理解了所有这一切，必须记住一件重要的事：这个项目仍然没做任何有用的事情，因为GUI还没有连接到事务逻辑层。现在着手排除这个障碍。

7.3.6 连接业务逻辑层到 GUI

在项目树中，右击TaskWindow.win项，选择“代码专家（Code Expert）菜单项，如图7.45所示（当TaskWindow.win在项目目录中被选中的时候，也可以使用Ctrl+W热键）。

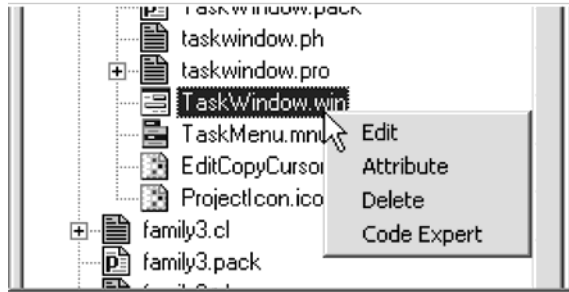


图 7.45 在项目树中右击 TaskWindow.win

在“代码专家”对话框中，确保id_file_open，id_Query_father，id_Query_grand_father和id_Query_ancestor_of 菜单项的默认处理程序已经设置，如图7.46所示。

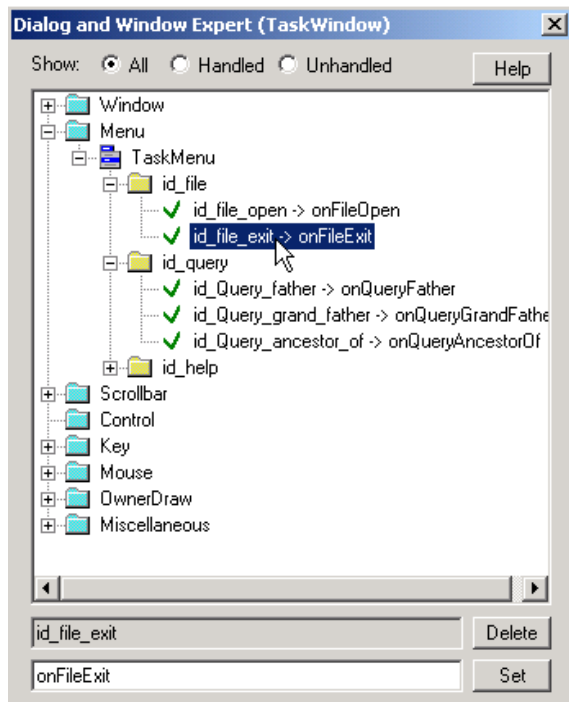


图 7.46 设置菜单的默认处理程序

正确退出应用程序的id_file_exit事件处理器已经被VDE设置，因此，不需要再次设定它。

在将正确的代码移植到处理器之前，显示层应该包含一个事实，该事实用来存储某时刻点程序处理的对象。

因此，在TaskWindow.pro文件中，必须给出下列代码：

facts

```
familyBL_db : (familyBL BL) determ.
```

上述事实将包含表示family的业务逻辑层的对象。

对于id_file_open菜单项，将会给出以下事件处理器代码：

```
onFileOpen(_MenuTag) = handled(0):-
    Filename = vpiCommonDialogs::getFileName(
        "*.txt",
        ["Family data files (*.txt)", "*.txt",
        "All files", "*..*"],
        "Load family database",
        [], ".", _),
    ! ,
    BL = familyBL::new(Filename),
    retractAll(familyBL_db(_)),
    assert(familyBL_db(BL)),
    stdIO::writef("Database % loaded\n", Filename).
onFileOpen(_MenuTag) = handled(0).
```

在编写另一个事件处理器之前，需要插入一个实用谓词，如下所示：

```
predicates
    tryGetFamilyBL:()->familyBL BL determ.
clauses
    tryGetFamilyBL() = BL :-
        familyBL_db(BL),
        !.
    tryGetFamilyBL() = _ :-
        stdIO::write("No family database loaded\n"),
        fail.
```

上述实用谓词用于处理这样一种情况，即用户在数据库被装入程序之前可能试图询问family数据库。正如前面的教程中所指出的那样，GUI允许依照用户的意图使用程序，程序员可以全然不知用户采取的事件执行顺序。实际上，允许用户决定在GUI程序中执行什么动作是一个好的风格。

因为在本教程中开发的程序遵从这一方法，因此在小程序中，当进行询问时，确信用户是否装载了数据库是不可能的。总有数据库不被装载的可能性。因此，若数据库没有被装入内存，程序被指定使用上述实用谓词来给用户一个适当的消息。

现在专注于其他的事件处理器。

对于id_Query_Father菜单项，将给出以下事件处理器代码：

```
onQueryFather(_MenuTag) = handled(0):-
    stdIO::write("\nfather test\n"),
    BL = tryGetFamilyBL(),
    BL.father(X,Y),
    stdIO::writef("% is the father of %\n", Y, X),
    fail.
onQueryFather(_MenuTag) = handled(0).
```

对于id_Query_Grand_Father菜单项目，将给出以下事件处理器代码：

```
onQueryGrandFather(_MenuTag) = handled(0):-
    stdIO::write("\ngrandFather test\n"),
    BL = tryGetFamilyBL(),
    BL:grandfather(X, Y),
    stdIO::writef("% is the grandfather of %\n", Y, X),
    fail.
onQueryGrandFather(_MenuTag) = handled(0) .
```

最后，对于id_Query_Ancessor，将给出以下事件处理器代码：

```
onQueryAncestorOf(_MenuTag) = handled(0):-
    X = ancestorDialog::getName(thisWin),
    stdIO::writef("\nancestor of % test\n", X),
    BL = tryGetFamilyBL(),
    BL:ancestor(X, Y),
    stdIO::writef("% is the ancestor of %\n", Y, X),
    fail.
onQueryAncestorOf(_MenuTag) = handled(0) .
```

观察上面的代码，注意到3个事件处理器都使用tryGetFamilyBL()实用谓词试图获得需要的对象，而不是直接访问事实段。在这种情况下数据库不被装载，实用谓词会在消息窗口给出一个适当的消息后失败。

7.3.7 理解事件处理程序

如果分析现在开发的事件处理程序，并且把它与那些在GUI中开发的处理程序做比较，就会注意到它们只有很小的不同。

最重要的是在第1个事件处理器(onFileOpen) 中。程序构造了一个新的程序事务层类的对象，并且将对象存入该模块的facts段。

要注意的另外一个关键点是业务层对象公共存取方法的使用，这些对象已经存储在TaskWindow.pro模块的facts段中。

例如，在下列代码片断中，程序获得一个业务逻辑层对象，并赋给变量BL，然后运行该对象的公共存取谓词grandfather。

```
...
BL = tryGetFamilyBL(),
BL:grandfather(X, Y),
stdIO::writef("% is the grandfather of %\n", Y, X),
fail.
```

变量和谓词之间的冒号算符“:”是一个逻辑指示器。获得对象公共存取谓词的冒号算符“:”和获得整个类的公共存取谓词的双冒号算符“::”不同。这可以在上面的代码片断

`stdIO::writef(...)`的谓词调用中看到。需要注意，双冒号算符不能用在对象上，而冒号算符不能用在类上。

7.3.8 运行代码

现在能够编译并且运行该程序了。如果有错误，那么应该重新阅读代码，或者也可以查看以前的内容去检查程序中的错误。需要注意的一点是，当设置事件处理器的时候，不要修改以前已经设定好的事件处理器的名字。

当运行这一程序时，会发现程序运行方式和以前的一样。这是面向对象程序设计的一个好的基础：它允许相当多的内部编码发生变化，但最终的运行功能并不发生变化。

7.3.9 细节考虑

在面向对象的程序设计系统中，类事实变成对象事实，因为每个对象将封装它自己的事实。

再次使用自助餐厅的例子，并且举一个与比萨饼上的调味酱相类似的例子，这种情形可以看作在每个比萨饼上的调味酱会伴随着每个比萨饼一起被送出。调味酱并不会被留在餐厅，并且顾客不会为了调味酱而重返餐厅。幸运的是，调味酱会伴随着比萨饼一起被送出。面向对象程序设计也是这种情形，它会把这些事实打包为一个类的对象，而不是把事实放在类中。

在使用对象表达数据时，数据的维护变得更加容易。如果不是用面向对象的方法，则可能需要用`retract`谓词（`retract`是Visual Prolog 6的一个内部谓词）清除剩余的早期数据。然而，现在不再需要`retract`，因为一个新对象建立的同时，旧的对象会自动被程序通过一个垃圾回收过程予以清除。这个操作在已知的安全条件下，在后台自动发生，不会影响用户。

声明谓词时，谓词的参数流模式必须要明确地添加到谓词声明中，就像在.i文件中所做的那样。`anyflow`流模式不能被用在全局实体上。

在事务层的执行代码中，执行了一个保存数据到一个文件的谓词。注意，这个保存数据的谓词并不需要文件名，这是因为文件名存储在对象中。下面作为一个练习，读者可以将这个功能与一些合适的GUI事件处理器结合起来。

7.3.10 小结

在本节中可知，业务逻辑及其表示层（GUI）的数据的较细分离不仅是可能的，而且是强烈推荐的。Visual Prolog 6能够智能化地创建一个包括所有类的程序包，这些类用于管理这些分离的事务。在这个特殊教程中，一个类被开发在一个程序包中，这种类可以创建封装数据和业务逻辑的对象。其他模块可以使用这些对象运行与业务逻辑层有关的一些内容。这样，系统维护和软件模块的开发变得更加容易，减少了出错的可能性。

7.4 Visual Prolog 的数据层

本节将继续以 `family` 为例介绍 Visual Prolog 能够更好地处理的下一个抽象层次——数据层。本节给出了处理复杂情况所需的各种组合。

这里提供的范例项目源文件，只能运行在 Visual Prolog 的商业版（commercial edition）环境下。

7.4.1 基本概念

中国有句谚语：“授人以鱼不如授人以渔”。这个谚语在编程当中同样适用。与其传授替代数据（鱼），不如传授通向数据的方法（渔）。

将这个概念引伸一下。例如，如果在传授一个人如何钓鱼以前告诉他从大自然中获取食物的法则，那么他就会有更多的机会为他的家庭获得食物。

但是有一个告诫：用类推解释事物可能产生一个很大的疑问，因为，当将类推推广得太远时，它就不成立了。在上面的例子中，如果给渔夫的概念再深入一层，那么在他能够享受教育的成果以前就会被饿死。

这种通过一系列相关行为而获得最终结果的方式同样可以在程序设计中使⤵用。但是这种方法需要巧妙地使用，要按照所需处理的软件问题的复杂性引起的要求而定。尽管有人说引用的层次越多，就越容易控制最终结果的实现，但是在设计层次以前还是需要仔细地考虑。

如果看到整个 `family` 系列(Visual Prolog 基本层、GUI 层、逻辑层)的展开模式，就会发现：随着引用越来越多的精确的层，程序的运行将得到越来越细微的控制。同时，不要有多余的层。否则它只能增加程序的复杂性和麻烦性。

本节将介绍数据层（data layer）。数据层的核心就是一个类，以及使用该类来访问和设置实际数据的对象。在上一节，业务逻辑层（business logical layer, BLL）也可以处理数据。而在这里，业务逻辑层 BLL 只处理逻辑。

7.4.2 程序

首先，根据本节所提供的文件 `family4.zip` 来了解程序的各个不同组成部分。在本节中，为了避免篇幅过于冗长，没有提供所需的代码。全部代码出现在所用的实例中。当安装实例项目时，可以从 VDE 读取代码。这里的部分代码只是用于快速参考。

当在 Visual Prolog 的 VDE 中安装项目 `family4` 时，就会发现 4 个独立的数据包，如图 7.47 所示。

这些程序包是 `FamilyBLL`，`FamilyData`，`FamilyDL` 和 `TaskWindow` 包。当创建项目时，VDE 会自动生成最后一个包（即 `TaskWindow` 包）。在前面的章节中已经介绍了其他包的生成方法。

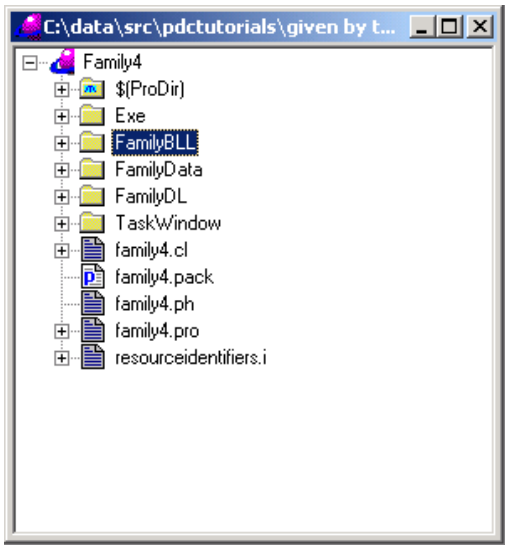


图 7.47 family4 的项目树

项目的创建与前面章节中介绍的创建方式类似，没有什么奇特之处，这里只对新增加的内容做一些解释。

与前面章节中所给出的例子相比，TaskWindow 程序包增加了一个新的对话框，即 NewPersonDialog，如图 7.48 所示。项目中已经包含了这个对话框，下面的解释假设要在项目中手动生成该对话框。

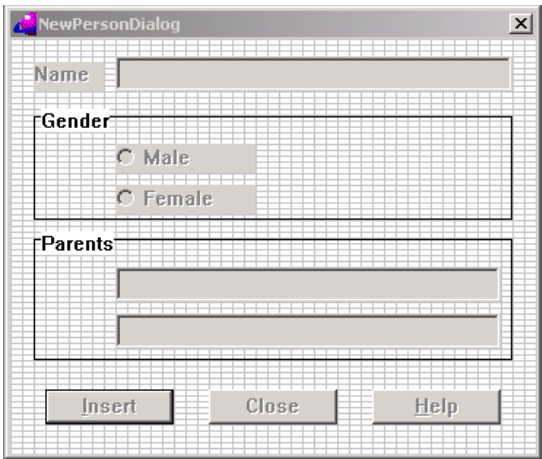


图 7.48 TaskWindow 新增对话框

该对话框是一个非模态（modeless）对话框，用于收集当前数据库新插入人员的信息。在图 7.48 中可以看到，有一个文本编辑框(idc_name)用于填写姓名，一组单选按钮(idc_male 与 idc_female)用于确定性别，两个长的文本编辑框(idc_parent1 与 idc_parent2)用于得到数据库新增人员父母的信息。非模态对话框可以保持开放和活动的状态，而不会妨碍 GUI 其他部分的工作。

模态与非模态对话框在创建上没有区别。只是在属性设置上要清楚地指明对话框的类型，如图 7.49 所示。

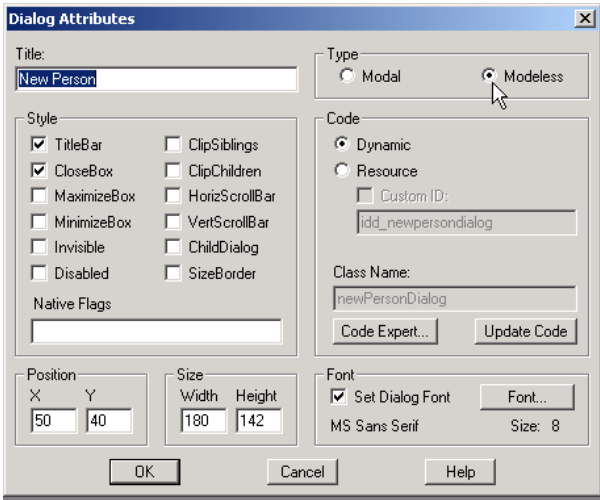


图 7.49 设置对话框类型

仅创建这样一个对话框是没用的，TaskWindow 必须能够调用它。这样就需要插入一个新的菜单项目和与之相关的事件处理程序。一旦创建了一个适当的菜单项目，代码专家将允许创建一个处理程序来添加必要的代码，如图 7.50 所示。

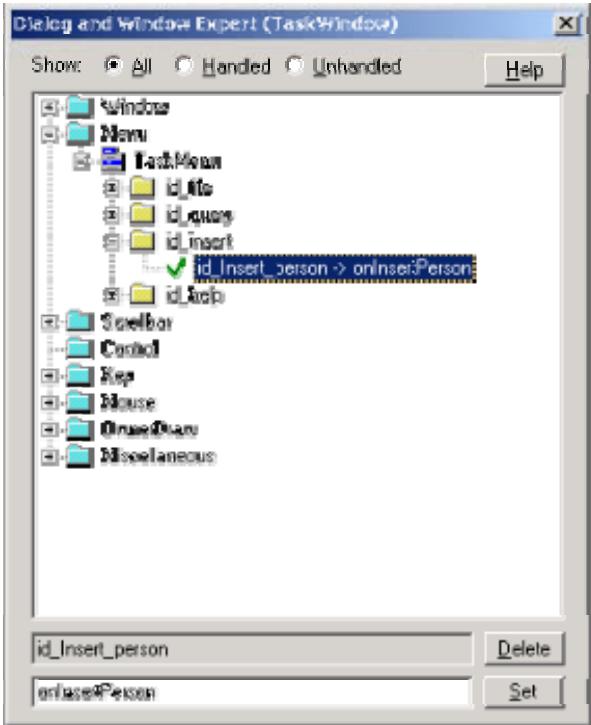


图 7.50 设置事件处理程序

注意：在前面的章节当中，已经解释了如何用上面的代码向导生成指定菜单项目的事件处理器。

7.4.3 非模态对话框

非模态对话框是一个概念问题。这个程序通过使用一个非模态对话框来收集 family 数据库中新增人员的信息。前面曾指出，对应于由非模态对话框引起的事件发生的代码有时具有欺骗性。如果了解模态对话框的工作方式就不难理解这一点了。在模态对话框中，操作系统暂停由同一应用程序中所有其他的 GUI 部分所引起的事件发生，只将注意力放在由模态对话框所引起的事件发生上。在时间逻辑上，程序员需要处理的内容就变得很简单了，因为程序员很清楚，与此同时不会有同一程序其他部分复杂事件发生。

另一方面，当使用一个非模态对话框时，程序不仅可以容纳由程序其他部分引起的事件发生，也可以容纳由对话框引起的事件发生。这样程序员就需要谨慎地考虑事件处理的排列（permutations）与组合（combinations）。例如，程序打开一个数据库，然后再打开一个该数据库下的一个非模态对话框。这样当数据库关闭而对话框没有关闭时，对话框的使用就有可能产生错误。当然，除非程序员已经预料到事件发生的过程并针对这种情况进行了设计。

在 NewPersonDialog.pro 中，如果看到了用于处理由对话框引起的事件发生的源代码，就会发现该代码捕捉到很多错误发生的情形。它甚至有一个完整性检查器用于跟踪可能导致一段代码产生错误的原因。

可能导致运行时间错误发生的谓词是从名为 trap 的一种特殊的内建谓词的内部进行调用的。例如，以下代码：

```
clauses
    onControlOK(_Ctrl,_CtrlType,_CtrlWin,
        _CtrlInfo) = handled(0):-
        Name = vpi::winGetText(vpi::winGetCtlHandle(
            thisWin, idc_name)),
        Name <> "",
        MaleBool = vpi::winIsChecked(vpi::winGetCtlHandle(
            thisWin, idc_male)),
        Gender = maleFlagToGender(MaleBool),
        OptParent1 = optParent(idc_parent1),
        OptParent2 = optParent(idc_parent2),
        trap(
            bl:addPerson(Name, Gender, OptParent1, OptParent2),
            TraceId,
            integrityHandler(TraceId)),
        !,
        stdIO::write("Inserted ", Name, "\n").
    onControlOK(_Ctrl, _CtrlType, _CtrlWin,
        _CtrlInfo) = handled(0).
```

在上面的代码片断中，程序员不清楚在数据库中添加一个新人员是否可行（因为不能够添加已经存在的人员）。这样就不能直接调用 `addPerson`，而是经由 `trap` 谓词调用。`trap` 谓词认可 3 个变量：第 1 个谓词用来完成代码中的实际工作；第 2 个谓词用于计数错误触发的次数；第 3 个谓词在错误触发的时候被调用。程序员可以通过第 3 个谓词将程序送回到安全的控制当中。

7.4.4 FamilyData 包

在这个例子当中，将创建一个 `FamilyData` 包，它包含了用于处理程序所需论域的所有必需代码。当业务逻辑层与数据层通信时，相同的论域可以访问 `BLL`。

在这个类中，编写如下代码：

```
class familyData
  open core

  domains
    gender = female(); male().

  domains
    person = string.
    person_list = person*.

  domains
    optionalPerson = noPerson(); person(person Person).

  predicates
    toGender : (string GenderString)
    -> gender Gender.

  predicates
    toGenderString : (gender Gender)
    -> string GenderString.

  predicates
    classInfo : core::classInfo.
end class familyData
```

注意：上面的类中包含了对多个类可共享的论域的定义。因此，`FamilyData` 包担当了数据层和业务层之间会议厅的作用。

7.4.5 接口

处理数据层需要经过由 `FamilyDL` 包中的一个类所产生的对象。然而，在获得该包的

实际内容以前，首先需要了解接口的概念。

接口可以看作是对与之相应对象的期望意图的描述。通过 VDE 可以将它写在一个单独的文件中，并包含使其能够在类对象中找得到的谓词。

接口也可以看作是对本节中所定义的进一步解释。读者将会在后面的内容学到有关这些接口的高级用法。

将目的相对独立的内容写在一个不同的接口中，原因很充分：根据一个接口的声明，可以写出一个类。接口只对类对象所提供的功能做一个简要的定义。之所以说简要定义是因为它仅仅对对象的某些功能进行了表达。简而言之，类可以公开声称其对象行为与接口一致。

不同的类可执行同一个接口；每个类提供一个专门的、具体的（即非抽象的）执行。这大大减轻了代码维护的负担。稍后，当同一程序中出现更多的组合情况时，程序员可以系统地在这类接口上工作。

7.4.6 FamilyDL 包

这个例子程序还包含了另外一个程序包，即 FamilyDL 包。该程序包包含了主类，其对象用于处理实际的数据。

该程序包包含一个接口，即 familydl 接口。该接口的主要用途是定义由数据层提供的面向业务逻辑层的谓词。同样它在数据层中也起了很重要的作用。该接口建立在 Family-Data 包中论域的基础上。

```
interface familyDL
  open core, familyData

  predicates
    person_nd : (person Name) nondeterm (o).

  predicates
    gender : (person Name) -> gender Gender.

  predicates
    parent_nd : (
      person Person,
      person Parent)
    nondeterm (i,o).

  predicates
    addPerson : (person Person, gender Gender).

  predicates
    addParent : (person Person, person Parent).
```

```

predicates
    personExists:(person Person) determ.

predicates
    save:().
end interface familyDL

```

在上面的代码中，关键字 `open` 用来指明该接口能够使用 `core` 与 `familyData` 包中的声明。

这个程序包包含两个类：`family_exception` 和 `family_factfile`。其大部分工作由 `family_factfile` 实现。该文件包含了能生成“鱼”的代码（在本节开头提到的中国谚语中所说的鱼）。它将系统地构建程序所需的数据库，并且在构建过程中，检查可能出现的错误情形。一旦出现错误，它通过使用类 `family_exception` 中的谓词来通知出现错误。

7.4.7 FamilyDL 包的代码

`familyDL` 包有一个接口文件 `familydl.i`，它的代码在以前的章节中给出过。该包中也有两个 `.pro` 文件，其中包含了两个 `.cl` 类文件的实现。这两个类为 `familydl_exception` 和 `familydl_factfile`。本节中没有 `familydl_factfile` 的代码，而该代码可以通过检查下载文件 `family4.zip` 来得到。该文件中的谓词用于处理数据，这一点在前面已经提到过。

本节中也没有 `familydl_exception` 的代码，而该代码可以在文件 `family4.zip` 中看到。这些谓词包括了支持实用程序的谓词：当程序发现不正确数据或错误（也就是计算机术语里所谓的异常）时，实用程序谓词用于显示相应的错误信息。很多错误情形的发生是超出程序员控制范围的（例如，驱动门被打开就是一个无法预料的异常），但是有时候需要“制造”一个错误。也就是说，程序员能够故意引发（`raise`）一个错误（这里用的是计算机界的行话）。这时会发现，在代码中错误实用程序谓词以术语前缀 `raise_` 开头。

值得注意的是 `familydl_exception` 自身不包括处理异常的逻辑。当异常发生时，应用程序谓词允许程序显示纠正错误对话框。

下面的例子说明了程序员如何通过故意调用这些 `raise_` 谓词来告知一个错误的情形。

在 `familydl_factfile.pro` 中，能够找到用于添加父母双亲的谓词。在子句当中，首先运行两个检查操作，查看 `Person` 与 `Person` 的双亲 `Parent` 是否已经存在，然后再调用 `familyDL_exception::raise_personAlreadyExists`。这样就构造了一个异常，程序会出示一个针对这种情况的一个专门的错误对话框。幸运的是，这个专门的错误对话框不需要程序员构建，而是由 `Visual Prolog` 的库在程序链接时生成。程序员不必像生成其他对话框那样，专门生成一个错误对话框。

```

% note: the following is only partial
%      code from familydl_factfile.pro

clauses
    addParent(Person, Parent):-

```

```

        check_personExists(Person),
        check_personExists(Parent),
        assertz(parent_fct(Person, Parent)).

clauses
    personExists(Person):-
        person_fct(Person,_),
        !.

predicates
    check_personDoesNotExist:(string Person).
clauses
    check_personDoesNotExist(Person):-
        personExists(Person),
        !,
        familyDL_exception::raise_personAlreadyExists(
            classInfo, Person).
    check_personDoesNotExist(_).

    ...

```

7.4.8 FamilyBLL 包的代码

下面给出 familyBL.i 接口文件的代码。这与前面章节提到的业务层的接口十分相似。但是，它们有十分重要的区别：论域已经移动到程序包之外，在数据层和业务层之间共享。这是一个很显然的移动，因为在先前的介绍中，把数据层和业务层当作同一段代码。当将这两层分开时，仍然需要某一公共端口。通过该公共端口，这两层的代码可以互相对话，显然需要将这两层的代码使用的论域变成为两者的公共端口。

第2个注意到的变化是谓词不会有多个流程。一个谓词只能对它控制的参数准确地做一件事情。这使得程序偏离了传统的 Prolog，变得与其他语言编写的程序很相似。通常，这个策略会产生出更好的结果。因为它使人们读源代码的时候更容易理解程序。但是，先前有多个流程的方法确实会产生更短的源代码，而且经常成为那些用过传统 Prolog 编程人员的首选方法。

```

interface familyBL
    open core, familyData

    predicates
        personWithFather_nd : (
            person Person,
            person Father)
        nondeterm (o,o).

```

```

predicates
    personWithGrandfather_nd:(
        person Person,
        person GrandFather)
        nondeterm (o,o).

predicates
    ancestor_nd:(
        person Person,
        person Ancestor)
        nondeterm (i,o).

predicates
    addPerson:(
        person Person,
        gender Gender,
        optionalPerson Parent1,
        optionalPerson Parent2).

predicates
    save:().
end interface familyBL

```

这里没有给出 `familyBL` 类的源代码。一旦将所提供例子的代码装载到 VDE 中，读者就会读到这些代码。这里没有什么新奇的东西。它同在先前所介绍的业务逻辑层中执行的行为是一样的。但是，有两点重要的不同：一旦需要访问数据或者数据设置，都会用到 `familyDL` 类中的谓词。同时，检查这个类的谓词信息的正确性，当发现逻辑错误的时候，导致异常产生。

这个包还包含另外一个类：`familyBL_exception`。该类同数据层中的类相似。但是这次，它包含实用程序谓词，这个谓词在业务逻辑层工作时，用来构建一个异常。

7.4.9 数据层的特征

数据层的主要特点是，它在面向对象语言环境中实现了一组称为数据访问器的谓词。数据访问器把数据访问同底层执行分离开来。简言之，可以使用谓词设置或获得程序所使用的数据。实际的数据会保持私有，导致外部无法访问。

抛开隐藏属性的具体结构，数据访问器同样能提供统一的接口来访问数据的属性。

使用数据访问器的优点在于：在程序的其他部分所实现的算法能独立于数据层中原来所用的数据的有关知识。在以后的内容中，将会看到在保持原样的同时，数据能容易地移入到一个更健壮的系统，譬如通过 ODBC 层数据访问来使用微软的 Access 数据库。

这种方法几乎不存在任何缺陷。即使偶尔存在缺陷，它们都能被正确理解。可能存在的主要缺陷是，当编写数据层的时候，程序员在理解上需要达到两个复杂的层次。一个是

确定内部数据结构在数据层中保持私有（这种需要开发数据结构的概念在前面已经讨论过）。但是在此之后，程序员仍将必须花费时间设计数据访问器谓词，这样其他的层同数据层能平稳地匹配起来。当一组程序员一起工作的时候，数据访问器谓词不正确的设计通常能产生许多令人头疼的事情，尤其是当程序从一个简单的部分移入到一个复杂的部分中时。这些问题都将在以后的内容中看到。

7.4.10 小结

在这一节的程序中，使用了这样一个策略，“授之以渔，而不是授之以鱼”。这只是将程序的源代码巧妙地模块化，这样代码维护与扩展变得特别容易。我们把 7.3 节中的业务层分成两部分：一部分包含纯粹的业务逻辑，另一部分仅仅是处理数据。这要求随后去扩充程序，仅用其他形式的数据处理技术来替换数据层（例如，通过一个 ODBC 协议，或者通过因特网等重新获得数据）。这一节同样涵盖了异常处理，能够很容易地编写自己的异常处理谓词，然后通过它们向用户显示有用的错误信息。

本章小结

本章介绍 Visual Prolog 编程方面的知识，主要内容包括 Visual Prolog 编程基础知识、Visual Prolog 的 GUI 编程、Visual Prolog 的逻辑层编程、Visual Prolog 的数据层编程等。

本章包含了 Visual Prolog 编程知识和工程应用程序开发时十分重要的内容。

习题 7

1. 传统的 Prolog 与 Visual Prolog 6 之间的差别主要体现在哪些方面？
2. 从结构上讲，Visual Prolog 的程序主要包括哪些段？各段分别有什么功能？
3. 按照书中介绍的步骤，测试完整的例子程序 family1.prj6。
4. GUI 程序与控制台程序有何异同？
5. 什么叫模态对话框和非模态对话框？
6. 根据 7.2 节的内容，学习如何用 Visual Prolog 6 开发并运行一个 GUI 程序。学会菜单项的设置与相应事件处理程序的结合方法。最后实践 7.2 节给出的例子（family2.zip）。
7. 什么叫业务逻辑层？业务逻辑层及其表示层（GUI）数据的较细分离有什么意义？
8. 上机实践 7.3 节中将业务逻辑与表示层分离的方法（family3.zip）。
9. 将数据处理从业务逻辑层分离的目的是什么？业务逻辑层与数据层有何关系？在 Visual Prolog 的 VDE 中安装项目 family4，实践并研究增加数据层后的程序结构、组成上等的变化。

第 8 章 编写 CGI 程序

本章分为基础和提高两部分内容。基础部分是指编写 CGI 程序的基础知识，内容包括公共网关接口、CGI 程序及其测试，还包括 CGI 的程序功能分析及输入流分析等。提高部分包括编写与测试 CGI 应用程序，内容有信息传递方法、高级 CGI 应用程序、CGI 程序的候选方案、加速 CGI 应用程序、CGI 程序的客户端、使用 Javascript 对象、安全性问题，以及 CGI 程序测试方法实例等。

在阅读本章时，会碰到很多新的概念或术语，不过在本章中会很快得到解释。

8.1 概述

已经介绍了 Visual Prolog 6 中的基本编程知识。此处不再对诸如类、接口、对象等概念进行叙述，也不会介绍 Prolog 中的回溯、子句、谓词等概念。此处假设读者对这些概念是熟悉的。本章中的例子不涉及任何类及对象的创建，所以对于那些不熟悉面向对象语言的读者来说，应当比较易懂。

最后一个例子用 Javascript 处理客户端进程。它利用了 Javascript 的面向对象的特点。要阅读本章的内容，读者最好要熟悉 Javascript 的概念。要想详细了解 Javascript 的面向对象的特点，不妨到下面这个网站去浏览其内容：

<http://www.webreference.com/js>

本章中讲述的 3 个 CGI 应用程序的例子都在文件 cgitutorial.zip 中。如果计算机中没有安装网络服务器，可以用压缩文件中自带的 TinyWeb web server。

8.2 编写 CGI 程序基础

本节是用 Visual Prolog 6 编写 CGI 应用程序的基础篇，通过举例说明用 Visual Prolog 6 编写 CGI 应用程序的一些相关基础。

8.2.1 公共网关接口

CGI 是指公共网关接口，是由世界万维网组织协会推荐的一种网络服务器输入和输出信息流规范。

正如人们所知道的，HTML 浏览器（如 Internet Explorer，Netscape）是通过网络服务器发送和接收信息的。但是万维网的作用并不只是读取一些超链接 HTML 文件，还需要有交互。即数据要从浏览器流入和流出，其中包括用户输入到浏览器窗口的信息。网络服务

器自己并不能解决所有问题，那么它是如何处理这些交互信息的呢？

网络服务器只是一个最基础的部分，它可以输出 HTML 文件到浏览器，包括其他的可识别的 MIME 类型的文件。但它并不支持一些高级操作，比如进入数据库后台并返回数据库的请求给浏览器。为了加强网络服务器的功能，于是 CGI 就产生了。服务器和同一台主机上的 CGI 程序进行对话后，CGI 应用程序就可以替服务器完成它无法完成的任务。比如进入数据库后台或执行复杂的搜索任务。应用 CGI 的网络服务器通常要比那些不用 CGI 的服务器可以发送更多的信息。

1. 公共网关

“公共”这个词代表能被所有的网络服务器接收的协议。“网关”这个词表示获得信息的行为，就像通过一个门或关口进入另一端的程序进程一样。抽象地说，网络服务器是网关的一端，而 CGI 应用程序表示网关的另一端，它们通过 CGI 彼此进行对话。

2. 流

通常所说的术语流，是指信息处理时每次只能处理数据队列中的一个字节。就好像一根管道，它的直径只能允许通过一个字节，这些字节以流的形式从通道的另一端流出，每次只有一个字节。

正如水不能同时在水管中朝两个方向流动一样，程序中的字节流也不能同时进行读和写操作。数据或者从流中输出（通常被称作可读流），或者输入到流中（称作可写流），但在单个程序中不能对同一个流同时进行读写操作。

对 CGI 程序来说，字符是 ANSI 格式的，是 8 位的字符。这一点以后必须要注意，因为 Visual Prolog 默认的是 UNICODE 字符，它是 16 位字符。Visual Prolog 中有专门用来接收 ANSI 字符流的谓词，所以处理流并不是问题。最后需要指出的是，当浏览因特网的时候，数据流通过因特网通道流出到浏览器，一次只有一个字节。甚至注意不到字节的通过，因为一旦单个字符到达浏览器，它们很快被组合起来，所以看到的好像是那些字节同时到达。

3. MIME 类型

MIME (Multipurpose Internet Mail Extensions) 比较确切的中文名称为“多用途互联网邮件扩展”。它是当前广泛应用的一种电子邮件技术规范，基本内容定义于 RFC 2045-2049。自然，MIME 邮件就是符合 MIME 规范的电子邮件，或者说根据 MIME 规范编码而成的电子邮件。

在 MIME 出台之前，使用 RFC 822 只能发送基本的 ASCII 码文本信息，邮件内容如果要包括二进制文件、声音和动画等，则实现起来非常困难。MIME 提供了一种可以在邮件中附加多种不同编码文件的方法，弥补了原来的信息格式的不足。实际上不仅仅是邮件编码，现在 MIME 已经成为 HTTP 协议标准的一个部分。

MIME 的目标之一是在数据文件和用于显示和编辑该文件类型的应用程序之间建立联系。MIME 通过为每种数据文件提供一个内容类型来实现这一点。

在 Web 上，数据的交换要通过超文本传输协议 (HTTP)。Web 服务器和 Web 客户必须对文件的内容类型达成一致，并且客户端软件必须能够理解该种文件，或者能找到另外

的能使用该文件的应用程序。

MIME 类型是因特网的命名数据类型的一个标准。数据类型名由两部分组成，第 1 部分是说明数据的基本类型，如图像、视频、音频、文本等。由于文本有不同的类型，如 C 源程序、英文文本，以及保存图像有多种格式，所以数据类型的第 2 部分是用来说明它的特殊类型。`image/gif` 就是一个恰当的例子。第 1 部分 `images` 说明它是一个图像文件，第 2 部分 `gif` 说明它是以 GIF 格式保存的。

可以把 MIME 类型认为是文件的扩展名，它告诉 Web 服务器或客户能处理什么样的文件类型。大部分的 Web 服务器是根据文件的扩展名来决定文件的 MIME 类型的。当浏览器和一个 Web 服务器进行交互的时候，它首先会把它所能处理的 MIME 能力和所需要的资源通知 Web 服务器（特殊的 MIME 文件）。Web 服务器根据这些请求发送相应的文件给浏览器并由其进行显示，如 `html`，`jpg`，`gif` 等。Web 服务器不会给浏览器发送无法识别的文件，否则就以 ASCII 码或 HTML 的形式发送，这主要取决于 Web 服务器。

在信息以流的形式被发送到浏览器的同时，服务器会在流的开头加入报头，包括数据的 MIME 类型。图 8.1 就是一个典型的例子，它说明了从服务器发送到浏览器的信息的格式。这些文件到达浏览器的时候，浏览器首先分析它的报头，推断出数据表示的 MIME 类型，并根据它自己内部能够处理的 MIME 类型表，显示相应的信息。

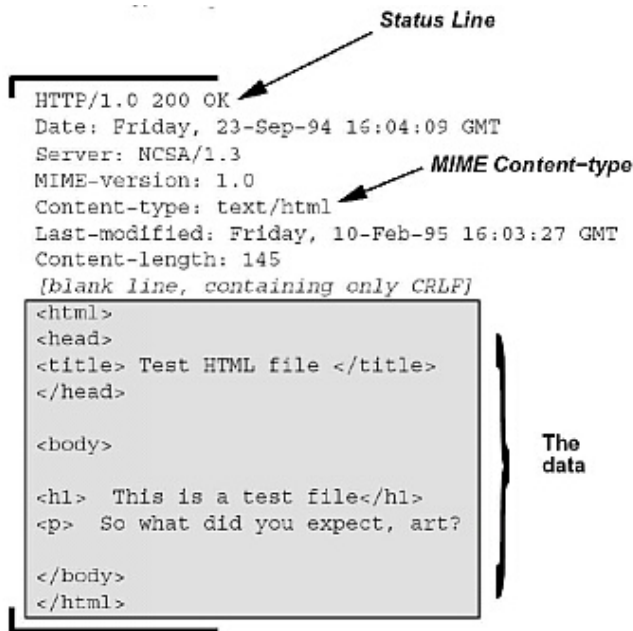


图 8.1 服务器到浏览器的信息格式

4. 报头

在这里，报头是指加在真正数据前的嵌入流中的一行文本，该文本后跟回车符。在上面的例子中，数据上面那一行就是服务器发送到浏览器的报头。之间用一行空白行把报头和正文分开。

例子中报头的内容类型 `Content-type` 这一行用来说明数据的 MIME 类型，这一行是专门应用于 CGI 程序的，因为当 Web 服务器调用一个 CGI 应用程序的服务来提供要传送给浏览器的信息时，它是忽略 CGI 程序提供的数据 MIME 类型的。所以，依照协议，CGI 程序必须在实际数据前加上报头（用一个空白行分开）。这一点在编写 CGI 应用程序的时候必须注意，因为不管是忘记了报头还是空白行（在实际数据和报头之间），都会带来很多麻烦。

8.2.2 CGI 程序

CGI 应用程序或网关程序是一个可执行程序（通常是 exe 文件），如图 8.2 所示，它并不通过键盘或鼠标接收任何信息。它并不是一个可交互程序（一个 CGI 程序应该是一个可交互的应用程序）。这种程序通常被称作控制程序，它通过标准 `stdin` 流接收输入信息，通过标准 `stdout` 流输出。报头的其他部分是在 CGI 应用程序完成其工作后由 Web 服务器加上的，然后这些编译好的信息被传送到浏览器。

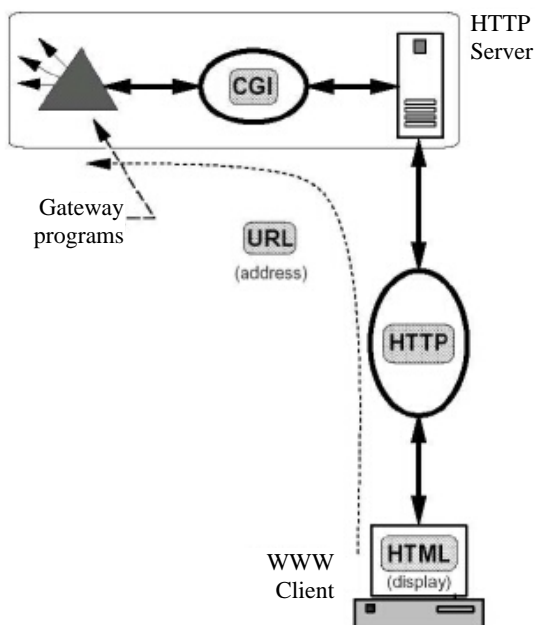


图 8.2 CGI 应用程序或网关程序

对 CGI 程序来说，一旦启动，CGI 应用程序在执行后会自动结束，而不需要任何输入来作用。这就像 DOS 中的 `xcopy` 和 `format` 命令一样，也是自动执行的。

CGI 程序不仅仅是一个控制台应用程序，它还等待接收符合规范的输入信息，并且输出信息也必须符合 CGI 规范。

在特定环境下，CGI 程序也可以写成 DLL 文件，但这种变更是依赖于 Web 服务器的。甚至也可以是其他可能的变化。比如，`bat` 文件也能做成 CGI 程序，前提是假设 Web 服务器被设置为支持该文件作为 CGI 应用程序。但不要陷入这些变更。在本章中，把 CGI

程序都认为是符合 CGI 规格的可执行控制台应用程序。

1. stdin 与 stdout

在很多程序语言中都有对 `stdin` 的定义，它是一个专门用来接收来自操作系统底层输入的流。程序（比如用 Visual Prolog 写的程序）可以读取 `stdin` 流并且将该流给出的字符作为程序的输入。同样，`stdout` 也是一个由操作系统控制的流，用来接收程序的输出信息。

不同程序的 `stdin` 和 `stdout` 流可以连接在一起。例如，A 程序写入信息到 `stdout`，然后调用程序 B，程序 B 用 `stdin` 流接收由程序 A 输出的信息。这就是网络服务器最基础的转换机制。在前面的例子中，A 就代表网络服务器，B 代表 CGI 应用程序，而 `stdin` 流和 `stdout` 流通常被放在一起称为控制台。

2. CGI 中的信息流

信息流是在客户向服务器发送请求时产生的。就像前边提到的，客户端会把它的处理能力和所请求的信息通知 Web 服务器。图 8.3 说明了浏览器（客户）向服务器发送的信息内容。

The following information is sent to the server by
the client (here by the Mosaic browser):

```
GET /Tests/file.html HTTP/1.0
Accept: text/plain
Accept: application/html
Accept: text/x-html
Accept: text/html
Accept: audio/*
.
.
.
Accept: */*
User-Agent:  NCSA Mosaic for the X Window
System/2.4  libwww/2.12 modified
[ a blank line, containing only CRLF ]
```

Request Header

图 8.3 浏览器（客户）向服务器发送的信息

第 1 行是方法域，用来指出所需的资源和 HTTP 方法，以及协议的版本等。在 `GET /Tests/file.html HTTP/1.0` 中，尽管 `/Test/file.html` 看起来像是路径，但通常称作统一资源定位符（Uniform Resource Locator，URL），它表达了很多内容，包括实际的文件名和文件在 Web 服务器上存放的路径。

还有其他通用方法，如 `POST`（客户向服务器发送数据（不是 URL 的一部分））方法和 `HEAD`（只接收资源的信息头）方法。其他域中是关于客户的一些信息。`Accept` 域告诉服务器客户所能接收的不同信息类型。信息是作为 MIME Content-type 类型发送给客户的，因此 `Accept:text/plain` 是指客户支持普通的 text 文件等。这些 MIME 类型很重要，因为服务器就是根据这个来告诉客户端发送数据的类型的。报头其他部分 `User-Agent:` 是发出请

求的程序；From: 是请求端；Referer: 给出发出请求的 URL 文档。服务器通过返回所需的数据作为回答。它首先发送应答报头，用来说明发送数据的状态，发送数据的类型和一些附加信息。后面是一行空白行，然后才是要发送给客户的实际数据。这些已经在上面的例子中做了说明。读者可能想知道，CGI 是怎么产生这些信息的呢？重新查看上面例子中所请求的资源，这些资源在 /Tests/file.html 或者是 /cgi-bin/helloworld.exe 中。许多网络服务器都把 URL/cgi-bin/ 设置为指向 CGI 程序 helloworld.exe，服务器必须能把放在这里的程序激活。

然后 Web 服务器将调用 helloworld.exe 程序，并通过环境变量和标准输出 stdout 流传送信息（对 helloworld.exe 来说是 stdin 流）给 helloworld.exe。然后服务器将等待，直到 CGI 程序 helloworld.exe 执行完毕。

一旦程序 helloworld.exe 停止执行，服务器将收集程序 helloworld.exe 写入的信息（包括 Content-type 报头），然后发送给等待着的浏览器。

所有这些会在极短的时间内发生。

现在，程序应该能顺利地按要求执行了。鉴于所有的 Web 服务器都能很好地完成这些工作，所以不必担心 Web 服务器激活一个 CGI 程序后该怎样工作，只需关注 CGI 程序是怎样从服务器读取数据的，并且执行完毕后又是怎样把数据传送给 Web 服务器即可。

3. 环境变量

所有的操作系统都允许程序向计算机的 RAM 中插入形如 name=value 的关系式，即所谓的环境变量。它是计算机 RAM（随机存储器）中的某个地方，如 DOS 中的路径设置就是一个环境变量。可以通过 SET 命令来查看 DOS（或 Windows）系统中的环境变量。只需在命令行提示符下输入 SET 命令，控制台就会显示所有可用的环境变量（在显示列表中也可以看到路径）。

环境变量的概念中最好的一个优点就是可以设置自己的变量并且为它们赋值。比如，如果想把密码保存在某个地方的话，就可以在操作系统命令行中用 SET 命令输入：

```
SET password=xyze
```

虽然值是字符串型的，但并不必给出引用。所有值都是以字符串的形式存储的，并且以字符串的形式被 Visual Prolog 程序取回。CGI 应用程序中环境变量的重要性在于服务器用各种环境变量来给 CGI 程序传送信息。下面是一个例子。

已经知道服务器在调用 CGI 应用程序之前把许多的数据放在 stdout 流里，但是 CGI 应用程序是如何知道要读入的数据有多少呢？为了能做到这一点，服务器把数据的长度赋给一个叫 CONTENT-LENGTH 的环境变量，CGI 程序通过这个变量的值得知要从流中读入的字符数量。分析 cgi.pro 中的 Visual Prolog 代码：

```
retrieve_POST_string() = Result :- % Get the length
    LenStr = environment::getVariable("CONTENT_LENGTH"),
    Len = toTerm(LenStr),
    % Read the stream to the desired length only
    Stream = console::getConsoleInputStream(),
    Stream:setMode(stream::ansi(ansi())),
```

```
% The stream is in ANSI mode!  
Result = Stream:readString(Len).
```

除了内容长度，许多 Web 服务器在启动一个 CGI 应用程序之前还会先设置如下的标准环境变量。有些服务器还可能对环境变量进行一些可能的变动，这主要取决于 Web 服务器。CGI 程序反过来也可以寻找到这些环境变量，并使用相应的变量值。限于篇幅，建议读者参阅一些关于环境变量的其他资料。比如，要想采取一些简单的安全措施，那么就可能需要用到环境变量 HTTP_REFERER，这一点将在后面介绍。

```
PATH_INFO  
PATH_TRANSLATED  
REMOTE_HOST  
REMOTE_ADDR  
GATEWAY_INTERFACE  
SCRIPT_NAME  
REQUEST_METHOD  
HTTP_ACCEPT  
HTTP_ACCEPT_CHARSET  
HTTP_ACCEPT_ENCODING  
HTTP_ACCEPT_LANGUAGE  
HTTP_FROM HTTP_HOST  
HTTP_REFERER  
HTTP_USER_AGENT  
HTTP_COOKIE  
QUERY_STRING  
SERVER_SOFTWARE  
SERVER_NAME  
SERVER_PROTOCOL  
SERVER_PORT  
CONTENT_TYPE  
CONTENT_LENGTH  
USER_NAME  
USER_PASSWORD  
AUTH_TYPE
```

8.2.3 测试 CGI 程序

为了测试 CGI 应用程序，需要有一个安装好的服务器，还要有 CGI 程序。还必须支持 HTML 文件，它通过网络服务器激活一个 CGI 程序。当然，这是随着程序的不同而不同的。学完这些例子后就会加深对所支持文件的了解。

前边已经讲过，CGI 程序必须和服务器在同一台计算机上。考虑到安全问题，服务器只能通过一个特定的路径（该路径称为 Web 路径）从主机上读取文件。所以，CGI 程序（CGI 可执行文件和 HTML 文件）可以放在这样的路径上，否则一些恶意用户可能会试着非法读取。

同样，网络服务器只能激活在正确路径上安装的 CGI 程序。CGI 程序不是放在网络上

的任何地方都可以的（有些服务器可以，但不是全部）。现在，所有服务器都有参数配置方法（一个独立的配置文件或.ini 文件或者 Windows 注册表）来设置各种参数。其中一个很重要的参数就是 CGI 程序的存放路径。当为 CGI 程序瞄准了 Web 服务器时，要知道的第 1 件事就是该 Web 服务器如何配置它的 CGI 应用程序路径。

1. 服务器的选择

如上所述，Web 服务器不只是一些复杂的软件，有许多免费的 Web 服务器可以选择。在 <http://www.serverwatch.com/stypes/index.php/d2Vi> 上就列出了很多。其中包括免费的软件和商业软件。Web 服务器的复杂性主要在于对 Web 路径的设置和对 Web 服务器的 CGI 程序路径的设置。

安装服务器时同时也要在计算机上安装 TCP/IP 协议（如果已经能浏览因特网，说明已经装好）。一旦成功安装服务器之后，就可以通过服务器的 IP 地址或机器的域名来获得服务器上的数据。现在的 CGI 程序能够与大部分的服务器一起很好地运行，包括微软的 Windows 2000 的 IIS5 Web server。

2. CGI 程序的存放位置

确保 CGI 程序存放在服务器的正确路径上。随着服务器的不同，这些存放路径也是不同的。本章认为是存放在路径 `http://localhost/cgi-bin/ <APPNAME>` 上，服务器上的浏览器可通过这个 URL 调用 CGI 程序。

比如一个名叫 `example1.exe` 的 CGI 程序，那么它在服务器上的 URL 应该是 `http://localhost/cgi-bin/example1.exe`，这样它才是可达的。

8.2.4 用 Visual Prolog 6 创建 CGI 程序

可以把 `cgitutorial.zip` 中的 `example.zip` 解压，然后对这个 Visual Prolog 例子进行测试。读者也可以根据下面的指导步骤自己重新创建一个。

（1）新建一个 Visual Prolog 项目

新建一个 Visual Prolog 项目，注意 Ui Strategy 中是 Console，如图 8.4 所示。

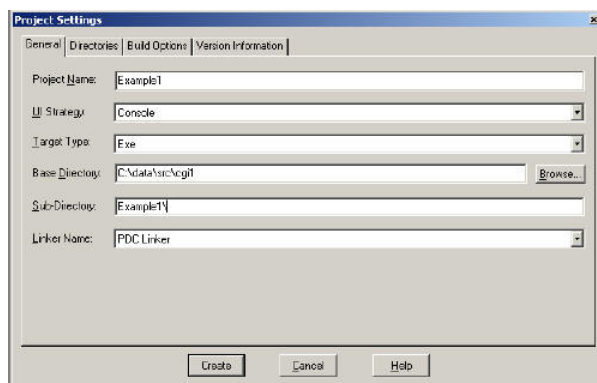


图 8.4 新建一个 Visual Prolog 项目

(2) 观察主项目树

这时就会显示出主项目树，其中包括由 VDE 决定的项目所需要的文件，如图 8.5 所示。

(3) 建立项目

打开 **Build** 菜单，选中 **Build** 菜单项或者按组合键 **Alt+F9**。随着建立过程的继续，VDE 会调用相关的 PFC（Prolog Foundation Classes）文档，这些文档在项目树中是看不到的。建立完毕后的项目树如图 8.6 所示。

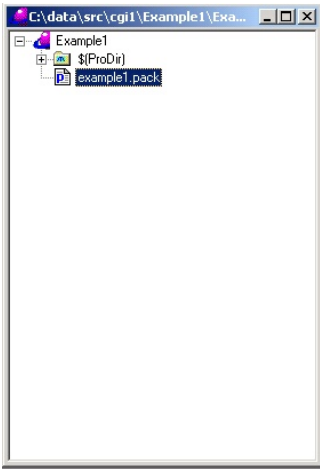


图 8.5 主项目树

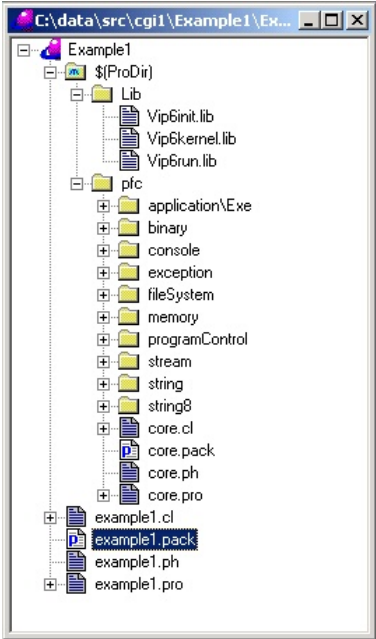


图 8.6 主项目树内容

现在已经完成了 95%的工作，这时在创建项目的 EXE 文件夹下将存在一个控制台应用程序，但它什么功能也没有。现在需要加入一些相关代码把它变成 CGI 程序。

前面已经讲过，CGI 程序要从 `stdin` 流接收数据（与环境变量一起），然后通过 `stdout` 流输出。由于这是一个很简单的 CGI 程序（第 1 个程序），所以将忽略从 `stdin` 流的所有输入（实际上服务器可能会将 `stdin` 流发送到 CGI 程序），但这里将向应用程序发送输出信息。

(4) 修改代码

双击项目树中的文件 `example1.pro`，并转向下方的代码：

```
clauses
    run() :-
        console::init(),
        succeed(). % place your own code here
end implement example1
goal
    mainExe::run(example1::run).
```

可以清楚地看到，程序的主要目标调用了被称作 `run` 的一个谓词，该谓词来自 PFC 模块 `mainEXE`。然后 PFC 再激活 `example1.pro` 中的 `run()` 谓词。分析下面所示的程序：

```
run():-
    console::init(),
    succeed(). % place your own code here
```

将上面的代码修改为如下的代码形式：

```
run():-
    console::init(),                                % Line 1
    OutputStream = console::getConsoleOutputStream(), % Line 2
    OutputStream:setMode(stream::ansi(core::ansi)), % Line 3
    stdIO::write("Content-type:  text/html\n"),      % Line 4
    stdIO::write("\n"),                              % Line 5
    stdIO::write(
        "<html><body><h1>Hello World!</h1></body></html>").
        % Line 6 (Don't forget the last dot!)
```

(5) 分析代码

现在逐行分析这些代码：

第 1 行：这一行是 Visual Prolog 必须要有的，所有的程序都要用到 `console`，都必须以 `console::init()` 开始。

第 2 行：获得 `stdout` 流对象，并将其赋给叫做 `Outstream` 的变量。前面讲过，CGI 程序希望发送它的输出到 `stdout`。这一行执行后，程序的输出将自动指向 `stdout`（所有的 `write` 语句都会写入到 `stdout`）。

第 3 行：在 Visual Prolog 中，默认的读写字符是 16 位 UNICODE 码，但这里需要的是 8 位的 ANSI，这一行就是用来把 16 位的 UNICODE 码转化成 8 位的 ANSI 码。

第 4 行：现在 CGI 程序将第 1 行传送给发出请求的服务器。正如前面所讲到的，这些数据必须要有一个头部。头部用来设置数据的 MIME 类型，这里声明的数据类型是 `text/html` 的 MIME 类型，注意这一行以换行符（`\n`）结束。

第 5 行：报头中必须要用一空白行把头部和正文分开，这是 CGI 规范强制要求的。所以 CGI 程序发送一个空白行（`\n`）。在 CGI 程序中经常会忘掉这个要求，这时可能会带来程序的错误执行。

第 6 行：最后真实数据被写入到 `stdout`。在这个例子中，程序仅是简单创建并发送了一个 HTML 文件，这个文件只包括两个字：“Hello World!”。

8.2.5 测试 example1

依照前边所讲的把它放在服务器的正确路径上，然后用合适的浏览器进入 URL `http://localhost/cgi-bin/example1.exe`。

8.2.6 应用程序功能分析

现在 CGI 程序 Hello World (example1) 已经不是原先的样子了。它很简单，但它可以充当很多复杂工作的基础。以最后一行为例，这一行完成对 HTML 页的创建并且发送 stdout 流传送给服务器，服务器再把该页发送到发出请求的浏览器上。如果建立了一个数据库，并在数据库后台加入一些处理数据库的谓词，就可以很轻松地建成一个管理系统。

8.2.7 输入流分析

就像前面所指出的，在这里忽略了服务器提供给程序的输入，如果 CGI 程序需要接收输入的话，就必须对 stdin 流进行读操作，然后对信息进行处理，而且必须检查好在 CGI 应用程序执行时设置的环境变量。这可能有些复杂，但幸运的是，在 Visual Prolog 中创建 CGI 程序时根本不必做很多编程。例如，如果想知道 CGI 程序接收了什么数据，只需用谓词 `cgi::getString()` 就可以了（在项目中，必须把 PFC 中的程序包 `cgi.pack` 包括在内）。如果想以更方便的形式得到数据，可以用 `cgi::getParamList()` 谓词。一个个的数据被谓词 `cgi::getParamList()` 组合为与环境变量相似的名值对。

8.3 编写实用的 CGI 应用程序

本节是用 Visual Prolog 6 编写 CGI 应用程序的高级篇，通过进一步举例说明用 Visual Prolog 6 编写 CGI 应用程序的一些高级技巧。

8.3.1 将信息从 HTML 文件传输至 CGI 程序

在图 8.3 中，展示了由浏览器向服务器发送信息流的详细资料。大部分全球信息网的实践由发送这类请求的浏览器组成。网络服务器依次处理这些请求，同时把需要的 MIME 类型数据送给浏览器。问题到那里还未完结。在 HTML 中，能利用 HTML FORM 的元素嵌入特殊的交互窗体。在 FORM 元素内，可以使用像 INPUT, TEXTAREA 等的交互式窗体。进入这种窗体的数据可以被分发到网络服务器。以这种形式在窗体中指定下列内容，如图 8.7 所示。

- 网络服务器对窗体信息应采取何种措施；
- 应使用哪种方法来实现该处理。

其中，ACTION 语句的后面只是一个 URL，它指定被网络服务器调用的 CGI 应用程序。在上图给出的例子中，它是一个被称为 `prgm.pl` 的 PERL 应用程序。数据是被网络服务器使用 GET 方法传给 `prgm.pl` 的。

```

<HTML> <HEAD>
<TITLE> Simple Form Example </TITLE>
</HEAD> <BODY>

<H3> FORM Example:</H2>
<FORM ACTION="http://name.ca/cgi-bin/prgm.pl"
METHOD="GET">
Type the time: <INPUT TYPE="text" NAME="time"
SIZE=30> <BR> Select Database:

<SELECT NAME="data base">
  <OPTION VALUE="harvard"> Harvard On-line
  <OPTION VALUE="toronto"> Univ. of Toronto
  <OPTION VALUE="seattle" selected> Univ of
Washington
</SELECT>
<BR>Keywords:
<INPUT TYPE="text" NAME="srch" SIZE=40>
<HR>
<INPUT TYPE="submit" NAME="submit" VALUE="Start
Search">
</BODY> </HTML>

```

URL

HTTP method (GET or POST)

图 8.7 窗体信息

1. GET 方法

在一个 GET 程序中，传递给 CGI 程序的全部数据在从浏览器被送到网络服务器的时候是被组合进 URL 内。假如要传送 3 个变量（名字、姓和电子邮件地址）及它们的数值到被称为 myemailer.exe 的 CGI 应用程序，这时 URL 可以写作

```
http://localhost/cgi-bin/myemailer.exe?firstname=sabu&lastname=francis&
email=sabu@somewhere.com
```

当汇编这些 URL 的时候，非法的字符要首先被转换到它们的 URL 编码形式。例如，如果任意一个数值包含一个空格，它必须被记为 %20，而一个冒号(:)将会被记为 %3A。这种编码形式叫做 URL 编码。在变量名和它的值之间利用间隔符号“=”分离。符号(&)被用于将一个名字数值对与其他的名字数值对区分开来。一旦 URL 到达网络服务器，CGI 程序将被触发，类似于本例中的 myemailer.exe，而且名字数值对的整个列表经由一个叫做 QUERY_STRING 的环境变量向前传递至 CGI 应用程序，它给出了在网络服务器与 CGI 应用程序通信的同时，服务器使用的环境变量列表。QUERY_STRING 的值将会是在“?”之后出现的任何值。在上例中，它应该是

```
firstname=sabu&lastname=francis&email=sabu@somewhere.com
```

2. POST 方法

在 POST 方法中，使用各种交互式 HTML 元素将数值传送给 Web 服务器，这些交互式 HTML 指定了 HTML FORM 元素的有效子元素（例如，INPUT，TEXTAREA 等）。这些数值被网络服务器接收，并由服务器使用 stdout 流传递至指定的 CGI 程序。CGI 程序依序读它的 stdin 流，以获取这些流包含的变量和数值。正如前面所解释的，网络服务器的 stdout 流是可读的，类似于被网络服务器调用的 CGI 应用程序的 stdin 流。

注意，即使在 POST 方法中，定义 QUERY_STRING 也是可以的。例如，可以在应用程序名结尾的“?”之后定义一些参量，就像在前面所举的例子中显示的一样。因此，CGI 程序既能从 stdin 中读取数据，也能从 QUERY_STRING 环境变量中读取数据。这一功能在最后一个例子中（在 cgitutorial.zip 中的 Example3.zip）被用于区分各种不同类型的 POST 处理过程。

3. GET 方法和 POST 方法比较

GET 方法是可标记的。这意味着整个 URL 包含了所有需要被送往 CGI 程序的数据。缺点是以这种方式传送数据有一个上限（大约 1000 字符），且无论什么数据在被送往 CGI 程序的过程中都是可见的。

POST 方法的优点是它能处理大量信息，并且数据在传送过程中不易被查看（除非使用探测工具）。如果使用一个可靠的服务器，浏览器也已经可靠连接到网络服务器，此时即使使用探测工具可能也无法显示正向网络服务器传送的数据。POST 方法的缺点是通常要在 HTML 中构造一个 HTML FORM 元素，并将这一 FORM 元素用正确的 HTML 交互式元素，比如 INPUT, TEXTAREA 等组装起来，以使 POST 方法正常工作。有的方法避免直接构建一个 FORM 元素，但是那将使 HTML 里面包含结构庞大的 Javascript 程式。

4. URL 编码

(1) 非 ASCII 字符(代码值>128) 经由它们的 URL 八进制编码%xx 进行编码。

(2) 不能识别的或特殊的 ASCII 字符将通过它们的 URL 码编码。这些字符包括：‘~!#\$%^&()+={}|[]\:"';<>?/, TAB，也就是除 @ * _ - 及 . (小数点) 5 个字符以外的所有字符。

(3) 空格符将被编码成正号 (+) 或者 %20。

8.3.2 解释信息流的高级 CGI 应用程序

可以将包含在 cgitutorial.zip 中的 example2.zip 文件解压到一个方便的目录中，并在此查找 Visual Prolog 文件。这个例子使用了一个叫做 blat 的实用小程序（一个免费的程序和帮助文件一起包含在 cgi-bin 目录下）邮寄表格的内容。可以根据下面的描述写出一个这样的应用程序。

(1) 可以按照在本章例子 example1 创建过程所描述的步骤执行。这些步骤执行结束后，就要执行一附加步骤，使 CGI 程序能够接收来自网络服务器的输入。如前面所指出的，需要求助包含在 Visual Prolog 中的 PFC (Prolog 基类)。右击项目树上的 PFC 文件夹，显示一个文件对话框。通过 PFC 文件夹进入 CGI 文件夹，单击 cgi.pack。此时项目树中也应该显示出 cgi.pack，然后重新构建程序。

以上步骤实现后，就有可能使用 PFC 谓词 cgi::getParamList() 获得所有经由网络服务器送至 CGI 程序的名字数值对。谓词智能化地剖析数据，不考虑数据被网络服务器传送时使用的是 POST 方法还是 GET 方法。

(2) 双击 example2.pro，增加以下谓词：

```

class predicates
    assembleParams:(namedValue_list List,string Seed) ->string procedure
        (i,i).
clauses
    assembleParams([],S)=S.
    assembleParams([H|T],OldSeed)=Result:-
        H=namedValue(N1,string(V1)),
        NewSeed=string::concatList([OldSeed,N1,"=",V1,"\n\n"]),
        !,
        Result=assembleParams(T,NewSeed).

```

(3) 将 run() 中的子句改变为如下的形式:

```

run():-
    console::init(),
    OutputStream = console::getConsoleOutputStream(),
    OutputStream:setMode(stream::ansi(core::ansi)),
    stdIO::write("Content-type: text/html\n\n"),
    doIt().

```

(4) 增加一个新的谓词 doIt(), 如下所示:

```

class predicates
    doIt:().
clauses
    doIt():-
        trap(file5x::file_str("blatparams.txt",BlatParams),_,fail),
        PList=cgi::getParamList(),
        S=assembleParams(PList," "),
        file5x::filenameunique("abc",Unq),
        file5x::file_str(Unq,S),
        BlatCmd=string::concatList(["blat.exe ",Unq," ",BlatParams]),
        platformSupport5x::system(BlatCmd).

    doIt():-
        stdIO::write("Could not find the file blatparams.txt",
            " in the cgi-bin directory. Create this file"),
        stdIO::write(" with just one line (NO carriage return at the end",
            " of the line!), as shown in the example below:<BR><BR>"),
        stdIO::write("<PRE>-to sabu@archsfa.com -server archsfa.com -f",
            " sabu@somewhere.com</PRE><BR><BR>"),
        stdIO::write(
            "In the above line, replace sabu@archsfa.com with the email address",
            " of the person to whom the form is to be sent to."),
        stdIO::write(" Replace archsfa.com with the SMTP address",
            " that can be used, and replace sabu@somewhere.com"),

```

```
stdIO::write(
    " with the 'From' address that is to be used for the transaction."
    "All these addresses MUST be valid.<BR><BR>"),
stdIO::write(
    " More parameters can be added to this file ",
    "(e.g.the subject to be used,etc.)if you understand the BLAT utility"),
!.
```

(5) 在程序中加入 `5xPlatformsupport.pack` 和 `5xFile.pack`。这一过程与在本章前面描述的过程一致。这两个程序包是编译 CGI 程序所必需的。

(6) 调用建立过程。这时会出现一些对话框，询问在编译时是否包含其他包。回答“是”就表示全包含。

(7) 检查并试运行。与 `example1` 不同，本例中的 `run()` 谓词非常小。它只输出标准的 MIME 类型 `text/html` 头和一个空白行，然后将控制权交给 `doIt()` 谓词。这是必需的，因为 `doIt()` 有两个子句，第 1 个子句体是完成主要工作的，第 2 个子句在找不到此项目需要的配置文件时被执行。`doIt()` 的第 1 行试图使用 `file5x:file_str/2` 谓词读取配置文件。如果找不到该文件，那么它将立刻失效并调用 `doIt()` 的第 2 个子句。`doIt()` 的第 2 和第 3 行收集输入变量及其数值的列表（经由网络服务器传送），再从列表产生一个长的字串。这一字串被存储在 `cgi-bin` 目录下的一个专门文件中，并且这一文件的内容将被 `blat` 实用程序电子邮箱。前面提到的配置文件 `blatparams.txt`，包含了文件内容被送达人的 E-mail 地址。

(8) 这一例子有很多优点。能在许多 CGI 程序环境中将这一程序作为通用的捕获应用程序，任意形式的输入都能被接收并邮寄给所选择的任何人。

8.3.3 信息从网络服务器到浏览器的传输

信息从 Web 服务器到浏览器的传输不总是以单一流发生。只有最小的 HTML 文件，如在前面的例子中给出的那样，从网络服务器到浏览器的传送才是发生在一个流里。大多数的 HTML 文件嵌有其他的 MIME 类型，例如，图像或 Javascript 代码。HTML 元素，比如 `<SCRIPT SRC="...">`，`` 等触发和网络服务器的特殊连接。为显示或使用这些特殊的 MIME 类型，浏览器与网络服务器建立单独的连接。这就是为什么浏览器用一种多变的方式分别建立页面的各个部分（例如，图像）的原因。如果需要，甚至可以为每条信息流分别编写 CGI 应用程序。例如，能分别编写传送 Javascript 代码给浏览器的 CGI 程序和用来传送图像的程序。在本章节最后，介绍线程化讨论板（threaded discussion board）例子的时候，这将会是很有用的。

但是没有必要写这么多的 CGI 程序以操作一个 HTML 文件中的各种不同的信息流。确实不需要为浏览器页面的每个信息流编写独立的 CGI 程序。实现方法有很多，在最后一例 `example3.zip` 中将会了解到，如何创建一个使用 `QUERY_STRING` 来区别被送往浏览器的各种不同类型信息流的 CGI 应用程序。开始时，谓词 `run()` 将检查 `QUERY_STRING` 并转向执行依赖传入数值的代码。因此，即使只写了一个可执行文件，因为执行动作按照 `QUERY_STRING` 给出的命令发生改变，所以它像是具有 5 个不同的 CGI 程序。

8.3.4 CGI 应用程序简评

CGI 应用程序几乎没有限度地扩充网络服务器的功能。惟一的缺点是 CGI 应用程序必须运行得非常快。如果它花太多时间处理并输出数据，网络服务器将会暂停应用程序，并忽略任何 CGI 应用程序在暂停之后可能提供的输出。

8.3.5 取代 CGI 程序的候选方案

已经有尝试去改变规范，比如使用管道和套接字（pipes and sockets）。更高级的网络服务器有允许服务器本身扩展处理 CGI 流量的内建 API。这些扩展通常记为 DLL 文件形式。这些措施是为了达到较高的速度。然而也带来了兼容性问题。

幸运的是，核心 CGI 规范事实上仍被所有网络服务器所遵守。因此，如果抵制住研究改变 CGI 的诱惑，并且坚持 CGI 主规范，那么将能写出可在各种网络服务器上使用的 CGI 程序。

8.3.6 加速 CGI 应用程序

cgitutorial.zip 中的 example1.zip 和 example2.zip 例子完成了网络服务器端所做的所有数据处理，包括数据的描述信息。通过描述信息，实际上可以在客户端的浏览器窗口中查询所有 HTML 信息。在网络服务器端集成全部的 HTML 时常耗费大量时间。还有另一个缺点是：已集成的 HTML 不易再改变，除非重新编译 CGI 程序本身。要避免这一缺点的一个非常简单的方法就是，以 Javascript 的形式送原始的数据元到浏览器，并且让浏览器以最后显示的 HTML 形式，实际执行所有的信息描述。

8.3.7 CGI 程序的客户端

CGI 程序能在 Visual Prolog 语言中开发，并有助于客户端处理吗？

在这样的方案中，有助于客户端处理可以使用 Javascript 的帮助。现在，Javascript 是一种非常稳定的面向对象的语言。一个鲜为人知的事实是 Javascript 和 Visual Prolog 实际上几乎使用相同的语法，而且这很可能被应用于客户端处理。

1. Visual Prolog 论域与 Javascript 对象之间的联系

Visual Prolog 的一个令人惊奇的方面是，它使用的论域可以被映射（也就是重建）在 Javascript 中，甚至嵌套结构也可能被重建。惟一的例外是谓词论域。Visual Prolog 和 Javascript 之间的这种独特连接很容易被开拓，如 Visual Prolog 定义了一个论域：

```
nameStr= nm(string NameOfPoster, string Email).
```

相同的论域在 Javascript 中可被表现为一个对象。这里是它的构造函数和一个用于返

回构造函数创建的对象的一个函数：

```
//注意：下面是一个名字为 jsobjs.js 的 Javascript 文件
// the constructor
function nml( NameOfPoster, Email)
{
    this.name=NameOfPoster;
    this.email=Email; alert("Created a JS nml Object!");
}

//a function that creates an object using the above constructor
function nm(NameOfPoster,Email)
{
    return new nml(NameOfPoster,Email);
}
```

上述函数将被写入一个叫做 jsobjs.js 的文件。该方案（一个构造函数和一个函数）将被用于 Visual Prolog 中定义每个论域。下面阐述采用这一方案的理由。

假设有一个 Visual Prolog 谓词需要传送论域 nameStr 的一个约束变量给一个 Javascript 信息流，该怎么做？

```
class predicates
    returnNameStr:().
    writeNameStr:(nameStr TheName).

clauses
    returnNameStr():-
        V=nm("sabu","sabu@somewhere.com"),           % Line 1
        writeNameStr(V).                               % Line 2

    writeNameStr(V):-
        stdio::write("Content-type: text/javascript \n\n"), %Line 3
        stdio::write("V="),                             % Line 4
        stdio::write(V).                                 % Line 5
```

假定已经写了一个称为 myjs.exe 的已整合上述基本代码的 CGI 程序。这个程序的源代码没有完整给出。这些作为一个练习留给读者（提示：在 run()子句体中调用谓词 returnNameStr(), 参见 example1）。

下面编写一个小的 HTML 文件，如下所示：

```
<html>
<script src="jsobjs.js"></script>
<script src="/cgi-bin/myjs.exe"></script>
<body>
</body>
</html>
```

可以发现，即使实际上是一个空白的 HTML 文件，当把它装载入浏览器（使用 HTTP 协议）的时候，它会给出如图 8.8 所示的一个提示框。

从而，这成功地将 Visual Prolog 的论域转换为 Javascript 中的对象。可以要求 Javascript 以更易接受的方式描述实际内容，而不会显示像上面那样的提示框。



图 8.8 提示信息对话框

2. CGI 程序工作过程

主要过程开始于 `returnNameStr()`。第 1 行中，在论域 `nameStr` 里建立了一个变量 `V`。第 2 行中，使用另外一个谓词 `writeNameStr()` 将那个变量经由 `stdout` 输出。第 3 行是表头（在最后带有一个空白行）用于送 MIME 去请求浏览器。注意，这个例子与之前所有例子不同的是，MIME 是 `text/javascript` 而不是 `text/html`。理由是不像前面例子中的 HTML 输出，这一个 CGI 程序正在耗尽 Javascript 资源。

第 4 行只是输出字符串：“`V=`”意味着 Javascript 定义一个变量 `V`（注意，这是一个 Javascript 全局变量，而且选择 `V` 为它命名，与 `returnNameStr()` 中的 Visual Prolog 变量一样）。

第 5 行写出 Visual Prolog 的字符串变量 `V`。这就是不可思议之处：Visual Prolog 内部论域到字符串的转化确保 `namestr` 论域中约束变量的字符串表示正好来自 Javascript 指定的函数 `nm(...)`。现在将注意力转向使用 CGI 程序(`myjs.exe`)的 HTML 文件。开始的 `<SCRIPT>` 语句调用先前写好的 Javascript 文件 `jsobjs.js`，该文件包含了映射 Visual Prolog 域的 Javascript 对象。一旦 `jsobjs.js` 被浏览器解释，它将在内部产生被称作 `nm1` 的 Javascript 对象（注意，Javascript 对象叫做 `nm1` 而不是 `nm`。`nm` 是函数，`nm1` 是类）。HTML 文件的第 2 个语句 `<SCRIPT>` 使用 `SRC` 参数调用 CGI 应用程序。这个脚本为浏览器提供数据，使它有所反应。从效率上考虑，能在 HTML 文件中以下列 `SCRIPT` 块代替第 2 个 `SCRIPT` 语句。

```
<SCRIPT>
    V =nm( "sabu" , "sabu@somewhere.com" )
</SCRIPT>
```

因为那是经由 CGI 应用程序传送的。注意，Visual Prolog 使用算符 `nm` 以区别 `namestr` 论域，并调用 Javascript 中的函数 `nm(...)`。此时函数 `nm(...)` 产生类 `nm1` 的一个对象并将它指派给 Javascript 的全局变量 `V`。现在考虑文件 `jsobjs.js` 中 `nm1` 的构造函数，当创建该类的对象时，将会出现一警告对话框。最后，这是迄今为止客户端处理的一个最好例子。如果留意就会发现，CGI 应用程序实际上什么都不做，只是分派某一特定论域的约束变量到浏览器，而实际的处理发生在浏览器端，这时相同的约束变量被当作一个 Javascript 对象再次出现。

8.3.8 使用 Javascript 对象的高级 CGI 应用程序

关于如何在 Visual Prolog 语言中使用 Javascript 对象的高级 CGI 应用程序，将在本章的最后一例中进行说明，包括 `cgitutorial.zip` 在内的 `Example3.zip` 中包含了一个被称为

Discboard 的高级 Visual Prolog 6 项目。在把 example3.zip 的内容解压到一个适当的文件夹之后，将 discboard.prj6 载入 Visual Prolog VDE 并查阅它的源代码。这是一个可执行的例子，使人们方便有效地使用线程讨论板（threaded discussion board）。

与 example1.zip 和 example2.zip 提供的注释不同 example3.zip 中源代码左边的注释和这里给出的说明很好地对该例子做了解释。

1. 了解所需要的论域

在大部分高级应用程序中，第 1 步要做的是构建一个恰当的服务与问题的数据结构。这里使用的讨论黑板与 Matt Wright 构建的讨论黑板十分相似（可参见相关演示 <http://www.scriptarchive.com/demos/wwwboard/wwwboard.html>）。

首先介绍线程讨论板（threaded discussion board）的概念：它看起来好像包含一连串的 post。其中，一个 post 是一种进入讨论板的许可证。然而，在更进一步的讨论中，将发现它实际上是一个树状结构：每个 post 将有多重应答作为其子 post。应答和 post 均没有数目限制。

一个新的 post 在技术上即是一个线程 thread。在它累积 post 作为它的子 post 的同时，它能隔离出单独的树（当它及其子句有应答时，加入论域 infinitum），一连串这样的独立树形线程构成整个讨论板。

如果研究文件 discboard.cl 所描述的论域，thrd 论域描述了线程讨论板的本质。如果研究该论域的最后一个参数，事实上，就会发现 thrdLis 是一连串的 thrd 论域。因此，每个讨论板的主要 post 的整个树结构都是在那一个论域中。

当学习文件 discboard.cl 时，thrd 论域中的其余变元是自我解释的：它们描述有关单一 post 的其他问题，比如建立 post 的人名、他/她的电子邮件地址、post 的日期、时间和 IP 地址等。

2. 查看执行情况

CGI 应用程序的主要运行部分开始于谓词 execute()，位于模块 discboard.pro 的末端。它要做的第 1 件事就是测试 QUERY_STRING 环境变量，然后重新定位执行程序为 exec() 谓词的 5 个子句之一。根据 QUERY_STRING，程序将做出不同的响应。

因此，从实用的目的来看，这个 CGI 程序可以起到 5 种不同应用软件的作用。给出的 QUERY_STRING 指令可能是这 5 种之一：addpost, replypost, delpos, showtree 和 showpost（QUERY_STRING 与数据库文件名一起传送，也用于线程化讨论）。这 5 个功能中，showtree 使用 Javascript 对象，并且使用 MIME 类型的 text/javascript 将它们传输到浏览器。

其余部分在功能上是相当直观的，是维持讨论板的数据所必需的。例如，当新的 post 被创建时，一个 thrd 项被插入到讨论板数据库的一个特定链（“__THREADS__”）。当 post 是对一个较早的 post 的应答时，它将作为一个子 post 插入到数据库中已经存在的 thrd 项中。所有程序，返回 HTML 数据到浏览器，展现运算状态。为方便操作，4 个功能（addpost, replypost, delpost 和 showpost）分别在它自己的 discboard.pro 中编写，用自己的谓词类 class predicates 划分。当读取 discboard.pro 的代码时，一次集中于一个区块。

但是，showtree 需要做一些额外解释。

当 `showTree()` 谓词运行时，它实际操作的是接收数据库顶端的 `thrd posts` 列表，并发送相同的列表，类似于一个 Javascript 数组的函数调用。这一个数组被文件 `showposts.html` 接收，根据 `dboard.js` 中的 Javascript 代码，该文件可以正确地对它们进行分析并变为 Javascript 对象数组。

需要测试 `dboard.js` 中的 Javascript 代码以了解 Javascript 函数 `thr(...)` 如何产生一个叫做 `thrd1(...)` 的 Javascript 对象，以及这种函数数组如何结束创建 `thrd1(...)` 数组对象。Javascript 函数结束构造一个 Javascript 对象的基本方法与前面介绍的类似，一旦 Javascript 对象的 `thrd1` 数组就位，在 `dboard.js` 中将定义名叫 `wrtAll(...)` 的 Javascript 函数，该函数将使用数组中的信息来构成所需要的 HTML，这些 HTML 全部被显示在浏览器中（`Dboard.js` 与 `showposts.html` 是存在于网络文件夹中的文件）。

在理解这个 CGI 程序的源代码之后，需要了解客户端处理部分，因此要介绍必需的客户端文件。这些文件是控制客户端处理的 Javascript 文件 `dconfig.js` 和 `dboard.js`。`dconfig.js` 是一个简单的配置文件，它的参数可被改变以适应讨论板。`Dboard.js` 是主要的 Javascript 文件，它包含了所有与 Visual Prolog 源代码定义的 `thrd` 论域和其他论域直接符合的构造函数及函数定义。`Dboard.js` 中的函数将从 CGI 程序接收到的指令翻译为相应的 Javascript 对象，类似于本章前面的描述。

其次，还需要相应的 HTML 文件：`showposts.html`，`rep.html`，`postmsg.html`，`replymsg.html`，`delmsg.html`。最后 `dboard.html` 也是必需的，它集成所有的 HTML 文件于一个框架之内。文件 `postmsg.html`，`replymsg.html` 和 `delmsg.html` 包含用来做交互工作的表格。

当调用讨论板时，它在 `showposts.html` 里面启动工作。该文件看起来像一个空的 HTML 文件，但实际上是启动 CGI 程序的 `showtree` 命令的文件，而且导致 Javascript 函数数组定义的 Javascript 对象，产生主要的讨论板树。一旦讨论板的树正确地进入浏览器框架，它就做好了接收“点击”的准备。点一下任何的信息链接，将在框架底部显示一个正确信息。而且顶端框架（使用文件 `rep.html`）包含讨论板所需要的其余交互式按钮和表格。

`rep.html` 包含很少的表格，而且调用正确 HTML 表格的分发系统依赖于被调用的函数。例如，单击 `Add Post` 按钮将会使 `postmsg.html` 页面显示到一个单独的窗口中。所有的 HTML 和 JS 文件被充分注释，以便对它们所完成的功能都能自我说明。

8.3.9 安全性问题

在结束本章之前有一些注意事项：CGI 程序没有与网络服务器相同的限制。一个网络服务器只能在网络路径包含的范围内工作。除网络服务器的安装目录外没有其他目录可存取一个网络服务器。如果在网络路径外面保存私人数据，可能要相当确定它不会将信息泄漏到任何其他刺探性的浏览器。然而，一个 CGI 程序没有如此限制。它给出许多能力，但是如果某一能力不被正确约束时，那么 CGI 程序就可能会使大部分数据暴露在外部视窗中。

CGI 程序的另一个问题是：尽管写一个忽略运行环境的 CGI 应用程序是相当容易的，但是这样的 CGI 程序可能很容易地被其他人盗用而自己不知道。举例来说，本章的 3 个 CGI 程序发展就有此倾向。这可通过下面的例子说明。下列是 HTML 超级链接编码：

```
<a href="/cgi-bin/example1.exe">First example</a>
```

它能应用于 `example1.exe` 所在的相同服务器（访问命令 `http:// site1`）。如果那个 HTML 在 `site1` 上找到，那么单击此链接会像期望的那样输出 `Hello World`。现在把 HTML 代码放在另一个位置 `http:// site2` 上，这时，不复制 `example1.exe` 到 `site2` 的 `cgi-bin` 目录。正如预期的那样，当再单击这一链接时，它将像预期的一样没反应，因为 `example1.exe` 不是真正在 `site2` 上。现在盗链 `example1.exe`，使用以下 HTML 代码：

```
<a href="http://site1/cgi-bin/example1.exe">
  First Example on another server</a>
```

现在单击这个链接，这时它会正常显示，即使 HTML 代码在 `site2` 上，而且 `site2` 机器的任何地方都没有 `example1.exe`。

8.3.10 防止 CGI 程序被盗链

避免 CGI 程序被盜链的一个非常有效的方法是查询一个被称为 `HTTP_REFERER` 的环境变量。这个变量值包含 HTML 文件在运行点提交至 CGI 程序的相关信息。这能很容易地被分析网站判断是否有效，而且 CGI 程序可能因此开始。修改 `example1` 并分析如何能避免上面例子中的盗链。

```
run() :-
    console::init(),                                % Line 1
    OutputStream = console::getConsoleOutputStream(), % Line 2
    OutputStream:setMode(stream::ansi(core::ansi)), % Line 3
    stdIO::write("Content-type: text/html\n"),        % Line 4
    stdIO::write("\n"),                               % Line 5
    run2(), !.

class predicates
    run2:() .
    checksOut:() determ.

clauses
    run2() :-
        checksOut,
        stdIO::write("<html><body>",
                      "<h1>Hello World!</h1></body></html>"),
        !.      %Line 6a

    run2() :-
        stdIO::write("<html><body>",
                      "<h1>Access denied!</h1></body></html>"),
        !.      %Line 6b
```

```
checksOut() :-  
    Refr=environment::getVariable("HTTP_REFERER"),  
    string5x::concat("http://localhost",_,Refr),  
    !.
```

为使用上例，必须将 `string5x` 和 `cgi` 程序包括进 `example1` 项目内(在建立过程中，VDE 可能会提醒给该项目添加更多的程序包。提示出现时只需单击 **Yes to all** 即可)。在编译修改例子后，会发现只有当它经由 URL 为 `http://localhost` 的 HTML 页面被显示时，`example1.exe` 才会工作。奇妙之处发生在 `checksOut()`谓词中，由它找到环境变量 `HTTP_REFERER` 的值是什么。这一环境变量经由网络服务器传至 CGI 程序，而且它包含呼叫 CGI 程序网页的 URL。`checksOut()`谓词的第 2 行使用 `concat` 谓词查看调用页面是否的确来自网址 `http://localhost`。也可以通过使用 URL 为 `http://127.0.0.1` (127.0.0.1 是 localhost 的 IP 地址) 试图存取脚本来检查上例，这时将会发现脚本显示拒绝访问 (Access Denie!)。

8.3.11 小结

CGI 程序给了网络服务器一个有力的路由扩充功能。它能以适当速度运行，但是如果使用 Javascript 的客户端处理，大部分处理将可能被送交浏览器，留下 CGI 程序仅做很少的工作。CGI 程序不比特别设计的控制台应用程序运行得更快。

8.4 CGI 应用程序测试实例

在本章“用 Visual Prolog 6 编写 CGI 应用程序”中提到了 3 个例子，它们都在文件 `cgitutorial.zip` 中。要调试这些程序，计算机上需要安装一个网络服务器。

如果计算机中没有安装网络服务器，可以使用压缩文件中自带的 TinyWeb 网络服务器。

这并不是说本章离不开 TinyWeb 网络服务器。本章的主要目的是讨论如何用 Visual Prolog 6 编写 CGI 应用程序，教程自带的 TinyWeb web server 只是为了便于读者检查和调试例子。

8.4.1 安装 TinyWeb 网络服务器

TinyWeb 是 RIT 实验室发布的一款出色的免费网络服务器。它是最快的也许是免费网络服务器中最为严密的一种。TinyWeb 与其他文件是分离的。它没有也不需要独立的安装程序。

如果想调试 CGI 应用程序，一定要将它安装在一台端口为 80，且没有任何其他网络服务器运行的机器上 (最好是 Windows 2000 操作系统)。

在 `cgitutorial.zip` 中的 `tutorial.zip` 文件中，注意两个程序：一个名为 `tinder.exe`，另一个名为 `tiny.exe`。双击图标运行 `tinder.exe`，它控制 tinyWeb 网络服务器 (`tiny.exe`)，因为网络

服务器在端口 80 上运行，所以在试图运行 `tinder.exe` 之前，要确保没有别的网络服务器在这个端口上运行。在 TinyWeb 中无须完成任何工作，因为一切工作由 `tinder` 完成。`tinder` 在系统区运行（通常在工具栏的右下角，看起来像一个飞碟）。双击它将打开一个小的控制面板。如果有必要，可以在开始时将 `tinder` 设置为自动运行。也可以通过控制面板设置网络服务器运行的端口和 Web-root。

8.4.2 TinyWeb 的根目录

安装好 TinyWeb 之后，可以通过一个浏览器（如 Internet Explorer）将其打开，其根的 URL 为 `http://localhost`，然后进行程序测试。在浏览器中列出的文件应当在 TinyWeb 所属目录的子目录里，形如 `index.html`。

8.4.3 TinyWeb 的端口

端口是指一种特殊的 TCP/IP 手段，用于运行特殊的互联网协议（可用的互联网协议有很多，例如 POP，SMTP 等。万维网是运行于 HTTP 协议之上的）。HTTP 协议的默认端口是 80。如果在计算机上已经有一个网络服务器在这个端口运行，可以改变一下这个网络服务器的端口号，比如改成 81；如果想通过 TinyWeb 网络服务器进入 HTTP 协议，将需要给出如下地址，即 `http://localhost:81`。

8.4.4 调试例子程序

本教程中编译过的例子已经放在 Web 目录（现在阅读的这个文件也是在这个目录下）下的 `cgi-bin` 文件夹中。确认 TinyWeb 网络服务器正常运行之后，把地址 `http://localhost/` 输入到浏览器的地址栏中。按照浏览器中所列出的说明，就可以运行所有的例子了。

8.4.5 用其他网络服务器运行例子程序

用别的网络服务器（如微软的个人网络服务器，IIS 等）运行这些例子同样很容易。只要保证在分离相关文件时，被分离的网络路径是该网络服务器所用的网络根路径形式即可。同样要保证网络路径中的 `cgi-bin` 路径应当是能被接受的可执行脚本形式，并且可以通过路径 `URL:/cgi-bin/` 到达。

本章小结

本章分为基础和提高两部分内容。基础部分是指编写 CGI 程序的基础知识，内容包括公共网关接口、CGI 程序及其测试、CGI 的程序功能分析及输入流分析等。提高部分包括编写与测试 CGI 应用程序，内容有信息传递方法、高级 CGI 应用程序、CGI 程序的候选方

案、加速 CGI 应用程序、CGI 程序的客户端、使用 Javascript 对象、安全性问题，以及 CGI 程序测试方法实例等。

习题 8

1. 用 Visual Prolog 6 创建、建立、调试并运行一个 CGI 程序。
2. 把 cgitutorial.zip 中的 example.zip 解压，然后对这个 Visual Prolog 例子进行测试。
3. 完成 8.3.7 小节中留给读者的练习（即 8.3.7 小节已经写了一个称为 myjs.exe 的 CGI 程序。这个程序的源代码没有完整给出，请读者完成剩余的工作）。
4. 何谓“公共网关”？何谓“MIME 类型”？
5. 测试一个 CGI 程序时，如何选择服务器？
6. 在 CGI 程序中，stdin 和 stdout 是何含义？如何设置和访问环境变量？
7. 解释 CGI 程序中的“输入流”。
8. 何谓 get 方法、post 方法？URL 编码的含义是什么？
9. 如何理解 Visual Prolog 的论域与 Javascript 对象之间的联系？
10. 在编写一个 CGI 程序过程中，如何保证安全性？如何防止 CGI 程序被盗链？
11. 如何用 TinyWeb 网络服务器来调试一个 CGI 程序过程？

第9章 编码风格

本章介绍 Visual Prolog 6 的编码风格，包括基本元素、推荐格式、程序结构、程序设计语用学、存储管理，以及异常处理。这里描述的 Visual Prolog 程序的编码标准，是 Visual Prolog 系统本身的一部分。用户文档中的例子也是标准的，它们同样也代表了 Prolog 发展中心为用户推荐的编码标准。

9.1 基本元素

本节描述 Visual Prolog 程序的基本元素，包括关键字、半关键字、文字、标识符、常量、变量、谓词、论域、类和接口等。

9.1.1 关键字

关键字以小写字母表示。在有关资料中，关键字是以没有衬线的粗体字被编排的，例如 Arial，默认颜色为暗黄色。例如，

constants	predicates
domains	class
facts	interface

9.1.2 半关键字

Visual Prolog 使用了大量的标识以满足多样化的句法结构，这些词以小写字母书写（除了 C 调用约定写成 C 外），且一般是没有衬线的字体。这些半关键字依照它们的性质以两种不同的颜色显示。如果这个词表示一种选择，那么它显示为藏青色；如果它是一种结构词，那么它将以暗黄色显示。

erroneous	stdcall
failure	C
procedure	...
determ	language
nondeterm	as
multi	...

下面这个例子显示了颜色和字体。

predicates

```
myPredicate : (string Value)
  procedure (i) language stdcall as "_myP"
```

9.1.3 文字

文字显示为蓝色。例如，
“Hello world!”

9.1.4 标识符

标识符的一般格式可以由下面的 EBNF 语法来描述：

```
<Identifier> = <Prefix> <WordGroups> <Suffix>
<WordGroups> = <WordGroup> { '_' <WordGroup> }*
<WordGroup> = <Word> +
```

前缀和后缀被用来表示某种标识符，并将用来处理各种标识符之间的联系。这些词以大写字母书写，当然除了整个标识符的第 1 个字母必须小写以外。

所有变量以大写字母开始，而其他所有的标识符以小写字母开始。

在文件中，除了关键字，所有的标识符以衬线字体编排。例如，Times New Roman 字体。

9.1.5 常量

常量既没有前缀也没有后缀，它以小写字母开始。例如，
numberOfRows, pi, logErrorMsg

9.1.6 变量

变量也没有前缀和后缀。像前面提到的 Prolog 要求的那样，变量以大写字母开始。在程序文件中变量以绿色显示。例如，

X, File, OutputStream

9.1.7 谓词

谓词没有前缀。然而，try 可以用来表示一个谓词是确定性的，特别是它被用做从一个相应的程序描述中区分确定性谓词的描述，而后者将引起一个异常而不是失败。例如：

```
trySetValue : (integer Value) determ (i).
setValue : (integer Value) procedure (i).
```

除非为了避免混淆必须添加后缀，否则谓词是没有后缀的。在一些情况下，为避免混

淆，表 9.1 中的后缀应该被选用。

表 9.1 常用的谓词后缀

后缀	意义描述	后缀	意义描述
<code>_db</code>	数据库算符/谓词	<code>_fail</code>	失败
<code>_nd</code>	nondeterm/multi	<code>_det</code>	determ
<code>_err</code>	erroneous	<code>_multi</code>	multi

注意，一般来说，多重谓词应以 `_nd` 为后缀，但如果环境需要也可用 `_multi` 代之。例如，

`setWindowFont`

`member`

`member_nd`

`ganttBar_db`

9.1.8 论域

论域没有前缀，`_list` 被用做列表论域的后缀。在多数情况下，列表论域没有域名。例如，一个数据库记录是一个值的列表，但是记录是列表值首选的一个更好的论域名。注意论域以小写字母开始。这同样适用于论域，如字符串、整数等。例如，

`string`

`value`

`record`

`record_list`

9.1.9 类和接口

类和接口没有前缀。例如，

`string`

`inputFile`

`template`

`inputStream`

传统的 COM 接口以 I 开始，现在这个 I 被保留了下来，但变成了小写：

`iUnknown`

`iDispatch`

9.2 推荐格式

本节考虑程序代码的格式。通过格式化，可以表示折行（line breaking）、缩排（indentation）和对齐（alignment）。缩排指行开始处的空格的数量，对齐指非行首字符的排列结构。

9.2.1 折行

折行遵守如下规则：

- 一行通常不应超过 70 个字符。

- 外部句法结构总是在内部结构之前被断开。
- 不同谓词的子句至少用一个空行分开。
- 同一谓词的子句不应被一个空行分开。
- 一个段的关键字之前至少有一个空行。
- 截断（无论它看起来如何微小）本身应该单独占一行。
- 一个子句的头在一行。

9.2.2 缩排

通过缩排，实现在行首的空格数量。缩排遵守如下规则：

- 缩排可由相同的步骤实现（例如，4 个空格）。
- 如果一套括号的部件必须被分在几行中书写，那么在开括号后面必须立即插入一个折行，且对于开括号缩排增加一步（没有对齐）。

9.2.3 对齐

对齐指的是排列结构，这种结构要么不是一行的开始，要么是通过缩排规则被对齐成一行的开始。

对齐没有被使用。

9.2.4 空格字符

- 逗号后面的空格可以被省略。
- 在算符或列表里面，逗号后面的空格可以被省略。
- 在声明谓词、事实、常量中，“：”之前或之后的空格可以被省略。
- 括号前后没有空格，除非这个括号与一个被空格所包围着的记号相邻，例如，‘：-’和‘：’。

9.3 程序结构

在这一节里，将逐一描述程序的结构。

9.3.1 段

段关键字本身就在一行中。如果段关键字被缩排 n 步，那么结构就被缩排 $n+1$ 步。段与段之间必须至少有一个空行将其分开。

```
clauses
```

```
  p.
```

```
clauses
```

```
q.
```

9.3.2 类、接口及实现

开类和闭类本身在一行。闭类包含类标识符，类里面的段关键字比类本身多缩排一步。

```
class specialOutputFile : outputFile
    predicates
    ...
end class specialOutputFile
```

这同样适合其他类型（如接口等）。

9.3.3 谓词声明

谓词声明总是将一些名字作为变元，这些名称被格式化为变量。模式的声明是可选择的。

```
Predicates
```

```
increment : (integer X) -> integer Y.
bubbleSort : (integer_list Input) -> integer_list SortedList.
myPredicate_nd :
    (aVeryLongDomainName StrangeFirstParamanter,
     anotherLongDomainName PlainSecondParameter)
    nondeterm (i,o)
    dterm (i,i).
```

注意：在开括号和缩排正常加大之后折行或者所有的变元在一个单行或者每个变元分开在不同的行。

9.3.4 论域

算符的参数总是有名称的，这些名称被格式化为变元。如果论域声明被断开成几行，那么在等号标记后的是第1个断行。所有的算符在一个单行或每个算符单独在一行。一个算子的所有参数占据一个单行或每个参数占据一个单行。

```
Domains
```

```
value_list = value*.
value =
    int(integer Value);
    real(real Value);
    str(string Value).
aVeryLongDomainName =
    x(interger X); y(integer Y).
```

```
anotherLongDomainName =  
    aFunctorWithManyArguments(  
        integer X,  
        integer Y,  
        integer Z,  
        integer RedColourComponent,  
        integer BlueColourComponent,  
        integer GreenColourComponent).
```

9.3.5 子句

子句头本身在一行里。子句体中每个调用在一行中。如果子句头必须被断开，那么变元比子句头多缩排两步，否则变元和子句体将被缩排在同一位置。

```
clauses  
myPredicate(X, Y) :-  
    callNoOne(X, Y, Z),  
    !,  
    callNoTwo(Z, Y).  
myPredicate(X, Y) :-  
    callNoThree(X, Y).  
aPredicateWithManyArguments(FirstArgument, SecondArgument,  
    X, Y, Z, ErrorNo, ErrorMessage) :-  
    callNoOne(FirstArgument, SecondArgument, X, Y, Z,  
        ErrorNo, ErrorMessage),  
    callNoTwo(...),  
    ...
```

9.3.6 不确定性循环

在 Prolog 中常用的一种结构是在不确定性调用结果之上的循环。对这种结构则将循环体缩排为一个额外的层次。

```
clauses  
myPredicate() :-  
    member(X, [1,2,3,4,5,6])  
        doAction(X), % extra indentation in the "loop body"  
    fail.  
myPredicate().
```

9.3.7 Word 格式化代码

这里将使微软文字处理软件 Word 的格式化代码变得更容易。

已经构造了一个代码段风格，此外还有许多风格，如关键字（keyword）、参数（para-

meter)、变量 (var)、文字 (literal) 等各种风格。

这个段落代码风格有下面的性质：

- 它遵照了文本段落体的格式。这种格式在代码段下提供了合适的空格。
- 它避开了验证（例如，拼写检查）。
- 它缩排一定数量，并将默认的 Tab 键大小设置为同样的数量。这里用 0.63cm，因为它是其他段落格式中的默认尺寸。
- 它保持相同特性的行在一起，以便程序部件不会跨页。

关键字的字符风格将字体变成 Arial，将字体版面样式变成粗体，将颜色变成暗黄。变量风格将变量颜色变成绿色，文字风格将文字颜色变成蓝色。

在打印代码的时候，应首先选择代码段风格，并使用软折行 (Shift+newline) 键入代码，使用 Tab 键进行缩排(在 tools -> options ... -> edit 页面上重置 insert and backspace set left indent 选项)。

Timesaver (节省时间)：双击关键字之一并选择关键字字符的风格。关键字一直选中的同时双击格式绘图笔 (工具栏中的画笔)，接着单击所有的关键字一次，当所有的关键字被着色后，按下 Esc 键或再次单击格式绘图笔。

9.4 程序设计语用学

当用 Visual Prolog 6 编写程序时，Visual Prolog 最好的实践包括 Prolog 发展中心(PDC)使用的格式建议与忠告。

该描述打算被用在新的 Visual Prolog 代码中，而 PDC 不希望将已有的代码更新为下面这些标准。因此，在有的 Visual Prolog 发行版本中可能没有下面这些标准。

注意，这些指南包含负面例子，例如，有毛病或有错误风格的例子等。这些例子（或者负面部分）将以红色显示，有时甚至以粗体显示。

```
clauses
    somePredicate(...) :-
        caseTest(...),
        caseAction1(...),
        !. % This cut is in the wrong place
    somePredicate(...) :-
        caseAction2(...).
```

9.4.1 常规技巧

一个谓词的子句通常应该少于 20 行，否则，应该考虑引入辅助谓词来处理次要任务。有时一个谓词可能占据多于 20 行。例如，一事物有 50 个属性必须被设置，且每个设置需要一个调用。

使用完全合法的名称（例如，someClass::method），不管什么时候它都使程序变得更

清晰。

当与类名一起读时，类方法应是一个有意义的名称。在方法名中避免重复类名，例如，这种名称可以是

```
someClass::method
```

而不是

```
someClass::someClassMethod
```

一个谓词应该只做一件事。因此，如果一个谓词用来处理列表中的每个元素，那么考虑将它分成两个谓词，一个浏览这个列表，一个用来处理这些元素。这样做的优点是使谓词变得更简单，因此更容易纠正，更容易理解。

9.4.2 布尔值

如果某事件或真或假，那么应该只用一个布尔变量。程序员不应该用它去区分两个不同的事件，例如，左与右、水平与垂直。在这些情况下，程序员应该用具体的论域来代替。

```
domains
    direction = left(); right()
    orientation = horizontal(); vertical()
```

当一个逻辑变量以真条件命名时，此时一个变元表达了一个这样的事实：如果事件为真就公布，若为假就不公布。称这样的变元为可公布的。

9.4.3 截断

截断是一种谓词，它去掉了不确定性，例如，去掉了更进一步求解的可能性（名称由此得来）。

所截断的这种不确定性可分为两组：

- 截断是阻断回溯的可能性而转到当前谓词中紧接着的下一个子句。
- 截断是阻断对一个不确定性谓词调用的更进一步的解决方法。
- 除了上面所述的两种用法之外，截断再没有其他合理的用法。一旦理解了这些目的，将截断放在一个正确的位置将是一件很容易的事。
- 或者将截断放在不再需要回溯后续子句的位置。
- 或者将它放在一个不确定性谓词的调用之后。对此，有一个惟一的解决方案是非常重要的。

第1个目的可以由下面的例子解释：

```
clauses
    p(17, X) :-
        X > 13,
```



```

    !,
    q(X),
    ...,
    p(A, X) :-
    ...

```

在这个例子中，程序中测试 $X > 13$ 之后有一个截断。这是第 1 个合理使用截断的非常典型的理由：“子句实例对输入进行测试且在直接测试 $X > 13$ 之后，即可找到正确的答案”。

一般来说，这样的一个截断典型情况下是放在子句头的后面，要么放在紧挨子句头的一个测试后面。

第 2 个目的可以由下面的例子解释：

```

clauses
    firstMember(X, L) :-
        member(X, L),
        !.

```

在这个例子中，截断被“立即”放在一个不确定性谓词的后面，只对一个解决方案有兴趣。

上面两次突出了单词“立即”，这是因为在放置截断时这个关键字是“立即”的，即截断应尽可能早地被放置在子句中。

对于包含多于一个截断的子句，应抱怀疑态度，一个多于一个截断的子句常常暗示了一个语法错误或一个设计错误。

9.4.4 红色截断和绿色截断

通常不鼓励将截断表示为绿色，红色截断相当完美。

传统的 Prolog 体系中已定义了红色截断和绿色截断的概念。简单地说，一个绿色截断出现时改变了谓词的语义，而红色截断则没有。

显然，所有截断一个不确定性谓词的更进一步的解决方法的截断自然是红色的。因此，区分红色截断和绿色截断只有一个目的，即阻断回溯到下一个子句。

考虑下面子句：

```

clauses
    p(X) :-
        X > 0,
        !,
        ...
    p(X) :-
        X <= 0,
        ...

```

上面谓词中的截断是绿色的，因为如果移动这个截断，这段谓词仍以同样的方式运行。如果这个截断出现在第 1 个子句中，那么第 2 个子句的测试 $X \leq 0$ 其实是不需要的：

```

clauses
  p(X) :-
    X > 0,
    !,
    ...
  p(X) :-
    ...

```

然而如果没有这个测试，这个截断将变成红色。因为现在如果移动这个截断，谓词将以不同的方式运行。

绿色截断可能似乎是多余的，但是，事实上它们被用做删除多余的回溯点（主要是考虑性能的原因）。在 Visual Prolog 中，绿色截断可能也被用来使编译器信服某些谓词可能有特殊的模式，例如，`procedure`。

9.4.5 指派输入格式

用一个调度程序仅能处理简单的事情。用调度程序处理堵塞问题，这一问题有一个非常单调的结构，特别是有许多不同的事件一定发生。因此，逻辑上相关联的代码扩展了，且基本上没事可做的代码块彼此紧挨着。

为了避免逻辑上相关联的代码被放在归类为一种模式的操作谓词中，所以调度程序仅调用处理谓词。在这种方式里，调度程序仅仅是调度程序，相关联的代码被靠近放置。

- 如果一个谓词处理很多情况，那么保持每种情况简单化。
- 如果一个谓词也处理其他的案例，那么同一个案例中不要有很多的子句。

第1个规则比较容易理解，第2个规则可由下面的例子加以解释：

```

clauses
  qwerty(17, W, E, R, 13, Y) :-
    ..., % A
    !,
    ... % B
  qwerty(17, W, E, R, 13, Y) :-
    ..., % C
  qwerty(Q, W, E, R, 13, Y) :-
    ... % D
  ...

```

上面的子句表示了错误的代码格式，因为它对同一个输入格式有两个子句。如果这是谓词处理的惟一方式，那么这样就行了。但是在这种情况下，也有其他形式的子句。通常认为上面的谓词应该被重写，以便谓词子句 `qwerty` 的目的是为了规定输入的具体格式，而将其他工作留给子谓词。

```

clauses
  qwerty(17, W, E, R, 13, Y) :-
    !, % we have cased out, this is one of our cases

```

```

    qwerty_17_w_e_r_13_y(W, E, R, Y).
qwerty(Q, W, E, R, 13, Y) :-
    !, % we have cased out, this is one of our cases
    qwerty_q_w_e_r_13_y(Q, W, E, R, Y).
...
clauses
    qwerty_17_w_e_r_13_y(W, E, R, Y) :-
        ..., % A
        !,
        ... % B
    qwerty_17_w_e_r_13_y(W, E, R, Y) :-
        ... % C
clauses
    qwerty_q_w_e_r_13_y(Q, W, E, R, Y) :-
        ... % D

```

这些代码已经将 `qwerty` 谓词变成将各种输入组合包装起来的谓词。像上面解释的那样，对于每种输入仅能有单个谓词调用，这并不是本质。主要的一点是不能以同样的输入方式从一个子句回溯到另一个子句。

这个规则特别被用于事件管理者，在同一事件处理中，不应当用多个子句去关闭或做其他事情。一个事件处理器经常分布在几页中，因此，如果它不具有一个案例被单一子句处理这样一个规则，那么就必须要去查看所有的子句，以确保知道单个输入格式是怎样被处理的。

9.4.6 异常和错误处理

- 当一个异常或错误发生时引起一个 `exception::raise` 异常。
- 如果想处理这个异常，就必须先捕获这个异常。

最终使用的时候，如果想在异常情况下运行某些代码，那么继续这种异常。

9.4.7 内部错误和其他错误

应该区分内部错误，以及与工具、模型、类、单元等有关的错误。如果某一内部变量被破坏，那么它是内部错误。典型的内部错误的例子如下：

- 数据库中应定义的事实没有被定义。
- 一个谓词应是一段程序，但编译器并不认可通过加一个失败的子句构成的谓词程序。如果那个子句是可达到的，那么是基于这样一个假设，即以以前的子句不能失败（一个变量）是错误的。

内部错误应该为每个单元分配一个异常，不过应总是使用一个异常：一个用作用户错误，一个用作内部错误（每单元用一个）。

典型的用户错误如下：

- 如果一个下标超出了限制。
- 如果一个窗口操作是错误的。
- 如果以错误的顺序调用谓词。

以下是想要捕获出口的两个原因：

(1) 因为某人想要处理异常，比如说打开一个文件，得到一个出口，说明这个文件不存在。在这种情况下想捕获这个出口并显示一些错误信息，以说明这个文件不存在。当然如果这个异常不是某人想处理的异常中的一个，那么就必须继续这个异常。

(2) 因为不管谓词是否存在而想要做一些事情，一个典型的例子是：得到了一些做某事的锁，并打开了这把锁。在这里想要确保，如果某事退出的话，这把锁也被打开。因此我们最终使用它。

9.5 存储管理

本节介绍在 Visual Prolog 中是如何管理存储器的。

从 Visual Prolog 5 到 Visual Prolog 6，存储器处理做了彻底的改变，尤其是因为现在的堆是依靠垃圾回收箱管理的。

因而在本节中将介绍运行堆栈、G-堆栈、堆和垃圾回收箱。

9.5.1 存储器

Visual Prolog 6 程序使用 3 种存储数据的方法：运行堆栈、全局堆栈、堆。

所有这些存储器都由各自的机制维持，并有各自的用途。

9.5.2 运行堆栈

运行堆栈用于参数和局部变量。当引入谓词调用时，其参数首先被压到运行堆栈，然后进行谓词调用。返回地址也传入运行堆栈。

参数命令和返回地址的顺序都是依赖于调用约定的，调用约定通过使用 `language` 关键词进行声明。

当输入谓词时，它将在运行堆栈中为局部变量和一些数据分配空间。

所有收集的参数、返回地址、局部变量等，统称为堆栈帧。当谓词执行结束时，堆栈帧将被移去。

迄今为止，这种处理与多数其他的程序设计语言中的处理一样，但是在处理不确定性谓词时区别较大。

当谓词带激活的回溯点返回时，如果使用了回溯点，则需要堆栈帧。因此，当谓词带激活的回溯点返回时，堆栈帧将不再被移去。

如果回溯点出现在嵌套调用中，堆栈帧也将保存，因而一个回溯点可以维持一个完整的堆栈帧“链”存活。

如果回溯点被删除，那么相应的堆栈帧链也将被移动。除非能确认该链栈顶，否则相信，无论何时，截断都能删除堆栈帧的“顶部”链(当然也删除了回溯点)。

运行堆栈经常被简称栈，有时也称为堆(尤其是在其他的语言里没有回溯的概念，因而当控制返回给调用者时，不保留堆栈帧)。

9.5.3 尾部调用优化

习惯上尾部调用谓词被写成递归谓词，也就是说，谓词访问自身。例如，表处理。每次谓词访问它本身时，将创建一个新的堆栈帧，如果这个表特别长，那么相当数量的堆栈帧将被创建。

由于堆栈帧有一个固定的尺寸（一旦它被创建），这对表的最大长度做了限制。

Visual Prolog 6 与 Visual Prolog 的以前版本一样，执行一种最优化，通常被称作尾部调用优化。其思想是：在最后访问谓词后，如果当前的堆栈帧不需要它，那么在调用尾部谓词前，它将被删除。

在这种思想的实际处理中有很多详细的资料，但是重要的事情是尾调用比其他调用使用更多的有效栈。然而，如果在谓词中有一个回溯点，就需要堆栈帧的撤回。

9.5.4 运行栈耗尽

如果曾用完运行栈，最大的可能性是在递归谓词中或在递归谓词的链上。在这里，与尾调用一样，递归不会发生；或由于回溯点的原因，必须保留堆栈结构。

9.5.5 全局栈

简单的数据，如数字，是围绕运行堆栈帧简单地被复制，但是在列表被作为参数或作为返回结果传递时，复制长的列表将是非常昂贵的。因此像列表这样的大量数据将区别处理。大量数据既可以存储在所谓的全局栈中（常叫做 G-堆栈），也可以存储在堆中。堆将在下面进行讨论，这里将集中讨论 G-堆栈。

这样的大量数据被表示为数据本身的指针，正如上面描述的指针在堆栈帧被复制一样，实际的数据一般存放在同一位置。

一旦列表和其他算符值被创建，它们将被存储在 G-堆栈中，它们将常驻在那里，直到 G-堆栈被弹出。

在任何需要的时候 G-堆栈将像堆一样增长，当回溯时它们将被再次弹出。其基本原理是，如果谓词失败，就不再需要数据了。这里不进一步详细描述这个机理。

9.5.6 G-堆栈耗尽

避免 G-堆栈耗尽的惟一方法就是回溯点。这看起来也许有点笨拙，因为正常的回溯点是在不能找到问题的解时要做的事。

这可能确实是个问题：一种以此方式写出的程序，即一个一直仅仅寻求预期路径的程

序可能是因 G-堆栈耗尽而终止。

GUI 程序从事件处理器中获得一些帮助，因为这些程序总是因失败而终止，并因此而释放 G-堆栈。每当事件处理结束，G-堆栈被重置为处理程序开始时的大小。

了解一下典型的失败回路：

```
clauses
  ppp() :-
    generator (X),
    action (X),
    fail.
  ppp().
```

在执行之前和执行之后(由于失败引起的回溯)的 G-堆栈具有相同的大小。

与 Visual Prolog 5 相比，Visual Prolog 6 在堆中存放比 G-堆栈中更多的数据，因而 G-堆栈问题比在 Visual Prolog 5 中要少发生。

Visual Prolog 6 只在 G-堆栈中放置算符数据，字符串和对象总是放置在堆中。

9.5.7 堆和垃圾回收

堆被用于数据，这些数据必须使回溯存在。对于那些不能被移动的数据（有关更多的内容在下面叙述），基本上指的是事实库和对象。但是为了澄清下面的描述，其他数据也可以存储在堆中。

堆通过垃圾回收箱来管理。程序员从堆中分配存储器（隐含地而不是明确地），但是它将不再释放存储器（隐含地或明确地）。取而代之的垃圾回收箱，它不断地分析堆并且释放不再需要的存储器，即它收集垃圾。

这是管理堆的原则。当需要堆存储器的时候，可能发生下面 3 种情况之一：

- 垃圾回收箱有一个是可利用的适当的堆存储器段，该段被返回。
- 从操作系统中分配新的内存。
- 堆是回收的垃圾，而分配是重新尝试。

9.5.8 垃圾回收

Visual Prolog 6 使用传统的 Boehm-Demers-Weiser 垃圾回收箱。

考虑一段程序，这段程序只分配存储器，从不对其重新分配。

在某一时间程序要做一些分配，但是，不是所有这些分配都能在程序中实现。程序只能存取存储器的直接指针或间接指针。但是当程序退出子程序时，指针就无效。子程序中的变量和参数也不再存在。

程序能达到的存储器和数据被称作活数据，其余的是死存储器或垃圾。

垃圾回收箱的目的是到处查找垃圾并使它能再使用（为活数据）。

为了查找垃圾，垃圾回收箱负责查找和标记全部的活数据，在此过程中，所有没被标记的存储器分配都是垃圾。

为了标记活的数据，垃圾回收箱从所谓的根系集合（root set）开始，它是全局变量加上现有的局部变量和参数。所有从根系集合指向的存储器被标记，否则所有从标记的存储器指向的存储器被标记。当不再有存储器可以用这种方式被发现时，所有的活数据即被标记，而所有没被标记的存储器是垃圾。

在垃圾回收箱回收垃圾之前，它为垃圾运行 `finalizers`。`finalizers` 是一个可选的用户自定义的例程，这个例程仅在一段存储器被回收前执行。运行相应的 `finalizers` 后，垃圾被回收，这样存储器就能再使用了。

9.5.9 Finalizers

Visual Prolog 6 在对象上支持 `finalizers`：为了拥有 `finalizers`，在一个对象构造类中简单地用谓词 `finalizers/0` 编写子句。该谓词被隐含定义，而不能显式定义。

注意：以 `waves` 形式回收存储器是垃圾回收算法的自然特性，在一段时间，有可能没有回收什么，有可能回收了很多，所以 `finalizers` 以 `waves` 的形式运行。

应该记住，当存储器处于非活动状态时，`finalizers` 不是（必要地）立即执行。在垃圾回收期间，当垃圾回收箱回收存储器时，执行才开始发生。

9.5.10 数据在什么地方分配

上面提到运行堆栈保存参数和局部变量，但那只是部分正确的。实际上运行堆栈只保存数字、字符和指针，任何非数字或字符的东西被表示为实际数据的指针。实际的数据可存储在 G-堆栈或堆中。

同样也提到，G-堆栈在回溯点上被弹出，因而任何激活回溯的数据必须存储在堆中。当数据被声明为事实时，它必须激活回溯，并且该数据被从 G-堆栈复制到堆（如果它是新近被存储在 G-堆栈）。

数据从不以别的方式被复制，数据一旦在堆中，那么它将一直存在于堆中，直到它被垃圾回收站回收。

确实没必要拥有一个 G-堆栈，在 G-堆栈中被分配的空间同样可在堆中被分配。但 G-堆栈比堆有一些性能上的优势：分配和去分配相当快，因为它只是改变一个指针。换句话说，无论数据是否是真活着，分配在 G-堆栈中的数据必须停留在那里，直到回溯时释放它。这样 G-堆栈将常常包含一些不可回收的垃圾。

因为 G-堆栈既有优点又有缺点，故对什么样的数据放在 G-堆栈没有约定。实际上，甚至不能保证 G-堆栈将在未来继续存在。

9.5.11 堆中数据

对象始终存储在堆中，主要原因是对象不能被复制，即不能从 G-堆栈到堆进行复制。对象不仅是一个值，它还携带有变化的状态。如果对象被复制，将存在对象的两个版本，一个对象的改变不会在另一个对象中被观察到。因而从一开始，对象就被存储在堆中。

字符串是被存储在称为原子堆的堆的特殊部分。堆的这部分是非常有效的。但是它只能用于不包括任何指针的数据。之所以有效，是因为它在垃圾回收期间从来不扫描指针。当已经知道数据不包含指针时，就没有必要去扫描它们。

同样提到，存储在事实中的数据也存储在堆中。

9.5.12 多线程和存储

多线程程序中的每个线程执行独立于其他线程的谓词调用和回溯，因此每个线程有自己的堆和 G-堆栈。换句话说，只有一个堆被所有的线程所共享。

只有一个线程能存取运行堆栈和 G-堆栈，因此即使没有更进一步的同步机制，它也能被存取。换句话说，在堆中的数据能被所有的线程存取。Visual Prolog 不强迫存储机制有任何的同步。程序员必须保证以尽可能明智的方式存取数据。

除了事实库（包括对象中的那些）之外，Visual Prolog 中的数据是不可变的。不可变数据没有任何的同步问题，很多线程可以同时读取那些数据。

当两个（或更多）线程同时更新同一段数据时将出现问题。这样，如果两个或更多线程同时从同样的事实库声明或撤销事实时，可能会遇到问题。

声明和撤销例程的执行可以确保事实库有一个好的结构，从某种意义上说，事实数据库的后续使用将不会引起任何的违规存取。

但是，如果两个或更多的线程同时更新事实数据库，一些更新可能会被丢失。举例来说，如果两个事实被同时声明，这可能造成一个事实被插入数据库中，另一个事实被丢失。

在读取事实库的同时，更新它是合理的。

PFC 提供了同步方法，可将它用于同步事实更新。

9.6 异常处理

所谓异常是指一段程序背离其正常的执行路径。PFC 的异常包包含了捕获异常的谓词等，它们会出现在 Prolog 程序中。

本节叙述如下内容：

- (1) 如何处理异常。
- (2) 如何构造自己的异常。
- (3) 怎样继续另一个异常。

本节讨论的范例程序是 exception.zip，可以在网上下载。

同时还可以参见 Visual Prolog 的在线帮助里“Prolog 基类/异常”一节中关于异常的详细描述。

9.6.1 如何捕获异常

首先考虑这样一段程序，程序的任务是读取一个文本文件并把其内容打印到控制台。PFC 提供了 file::readString 谓词，它能完成上述任务。但是，有一些情况可能妨碍实现该

任务。例如，一个指定的文件名可能指向一个根本不存在的文件。当谓词 `file::readString` 无法完成任务时，它就产生一个异常。

Visual Prolog 提供了一个内置的谓词 `trap/3` 来捕获异常并对此进行处理。关于谓词 `trap/3` 的更详细的内容，可参见 Visual Prolog 的在线帮助中的相关主题（“Language Reference → Built-in Domains, Predicates and Constants → Predicates → trap” 主题）。

捕获一个异常的代码如下：

```
trap(MyTxtFileContent = file::readString(myFileName, _IsUnicode),
    ErrorCode, handleFileReadError(ErrorCode)),
```

最有趣的部分是谓词 `handleFileReadError` 的实现：

```
class predicates
    handleFileReadError : ( exception::traceID ErrorCode ) failure.
clauses
    handleFileReadError(ErrorCode):-
        Descriptor = exception::tryGetDescriptor(ErrorCode, fileSystem_api::cannotcreate),
        !, % file cannot be loaded
        exception::descriptor(_ErrorCode, % _ErrorCode is ErrorCode ,
            %just not necessary to insert extra check here
            _ClassInfo, % class information of the class, which raised the
            exception .
            _Exception, % actually it is fileSystem_api::cannotcreate,
            % but the parameter should not be compared by ' = ' .
            % See exceptionState::equals
            _Kind,% exception can be raised or continued
            ExtraInfo,
            _CursorPosition,
            % currently we know the position,
            % but sending dumps to developers requires positions
            _GMTTime, % the time of exception creation .
            ExceptionDescription) = Descriptor,
        FileName = core::mapLookUp(ExtraInfo,
            fileSystem_api::fileName_parameter, string("")),
        Reason = core::mapLookUp(ExtraInfo,
            common_exception::errorDescription_parameter, string("")),
        stdIO::write("Cannot load file due to: ",ExceptionDescription,
            "\nFileName: ", FileName,
            "\nReason: ", Reason ),
        exception::clear(ErrorCode),
        % it is necessary to clean exceptions when they are handled
        fail.
    handleFileReadError(ErrorCode):-
        isDebugMode = true,
```

```
!,
exceptionDump::dumpToStdOutput(ErrorCode),
    % dump to console for developer needs
exception::clearAll(),
    % clear all exceptions, as they are shown in the dump already .
fail.
handleFileReadError(ErrorCode):-
    exception::clear(ErrorCode),
    % program cannot handle the exception and it does not report about it .
fail.
```

当预期的 `fileSystem_api::cannotcreate` 异常出现的时候, 具有捕获和处理这一异常参数的异常处理谓词 `exception::tryGetDescriptor` 恰好正确地被调用。例子中, 使用控制台的输出来给出异常原因的解釋:

```
stdIO::write("Cannot load file due to: ",ExceptionDescription,
    "\nFileName: ", FileName,
    "\nReason: ", Reason ),
```

完整的例子程序可参见项目 `catchException\catchException.prj6`。

9.6.2 如何构造自己的异常

考虑一个检查特定文件的长度或大小的程序。PFC 提供了一个谓词 `file::getFileProperties` 来返回一个特定文件的长度。如果文件长度为 0, 可以通过谓词 `exception::raise` 来构造一个异常, 同时有必要创建一个谓词, 以此指定该异常。为此, 已经创建了 `myExceptionZeroFileSize`。构造一个异常的代码如下:

```
clauses
run():-
    console::init(),
    trap(file::getFileProperties(myFileName,
        _Attributes, Size, _Creation,
        _LastAccess, _LastChange),
        ErrorCode,
        handleFileGetPropertiesError(ErrorCode)),
    Size = unsigned64(0, 0), % file size is zero
    !,
    exception::raise(classInfo,
        myExceptionZeroFileSize,
        [namedValue(fileSystem_api::fileName_parameter,
            string(myFileName))]).
run().
```

谓词 `exception::raise` 的第 1 个参量是 `classInfo` 谓词, 它是由 VDE 为任何新创建的类而生成的。当然, 在一个异常产生时, 指明额外的信息是一个好的方法。例子中, 文件的

名字非常有用，因为零长度归属于所指定的文件。

谓词 `myExceptionZeroFileSize` 用以下模板生成：

```
clauses
  myExceptionZeroFileSize(
    classInfo,
    predicate_Name(),
    "File size cannot be zero").
```

这就是说，异常谓词的第 1 个参数和第 2 个参数通常分别是 `classInfo` 和 `predicate_Name()`，第 3 个参数包含该异常原因的一个文本解释。

完整的例子程序可参见项目 `raiseException\raiseException.prj6`。

9.6.3 如何继续另一个异常

考虑一个 DLL 程序，该程序读取一个文本文件，然后用一个参数返回其内容。如果异常出现了，那么 DLL 继续，同时补充一个关于 DLL 版本的额外信息。继续该异常的代码如下：

```
constants
  dllVersion = "1.20.0.0".

clauses
  loadFile(FileName) = FileContent :-
    trap(FileContent = file::readString(FileName, _IsUnicode),
      ErrorCode,
      exception::continue(ErrorCode,
        classInfo, continuedFromDll,
        [namedValue(version_Parameter, string(dllVersion))])).
```

谓词 `continuedFromDll` 声明如下：

```
predicates
  continuedFromDll : core::exception as "_ContinuedFromDll@12"
```

它从该 DLL 中输出。谓词 `continuedFromDll` 的实现由下面的模板产生（与前面的 `myExceptionZeroFileSize` 相同）：

```
clauses
  continuedFromDll(
    classInfo,
    predicate_Name(),
    "Exception continued from continueException.DLL").
```

当谓词 `exception::continue` 继续该异常的时候，会增加一个关于 DLL 版本（`dllVersion`）的额外信息。

完整的例子可参见项目 `continueException\continueException.prj6`。

下面介绍如何捕获这样一个继续的异常。其代码与上面讨论的代码类似。在这里，只集中考查二者不同的方面。

```
constants
  myFileName = "my.txt".

clauses
  run():-
    console::init(),
    initExceptionState(exception::getExceptionState()),
    trap(MyTxtFileContent = loadFile(myFileName),
        ErrorCode, handleFileReadError(ErrorCode)),
    !,
    stdIO::write("The content of ",myFileName,"is:\n",MyTxtFileContent).
  run().

class predicates
  handleFileReadError : ( exception::traceID ErrorCode )failure .

clauses
  handleFileReadError(ErrorCode):-
    DescriptorContinued = exception::tryGetDescriptor(ErrorCode, contin
uedFromDll),
    Descriptor = exception::tryGetDescriptor(ErrorCode, fileSystem_api::
cannotcreate),
    !, % file cannot be loaded
    exception::descriptor(_ErrorCodeContinued,
        _ClassInfoContinued,
        _ExceptionContinued,
        _KindContinued,
        ExtraInfoContinued,
        _CursorPositionContinued,
        _GMTTimeContinued,
        ExceptionDescriptionContinued) = DescriptorContinued,
    Version = core::mapLookUp(ExtraInfoContinued,
        version_Parameter, string("")),
    stdIO::write("Exception continued : ",ExceptionDescriptionContinued,
        "\nDllVersion: ", Version,"\n"),
    exception::descriptor(_ErrorCode,
        _ClassInfo,
        _Exception,
        _Kind,
        ExtraInfo,
        _CursorPosition,
        _GMTTime,
```

```
ExceptionDescription) = Descriptor,
FileName = core::mapLookup(ExtraInfo,
    fileSystem_api::fileName_parameter, string("")),
Reason = core::mapLookup(ExtraInfo,
    common_exception::errorDescription_parameter, string("")),
stdIO::write("Cannot load file due to: ",ExceptionDescription,
    "\nFileName: ", FileName,
    "\nReason: ", Reason ),
exception::clear(ErrorCode),
fail.
handleFileReadError(ErrorCode):-
    isDebugMode = true,
    !,
    exceptionDump::dumpToStdOutput(ErrorCode),
    exception::clearAll(),
    fail.
handleFileReadError(ErrorCode):-
    exception::clear(ErrorCode),
    fail.
```

在该段代码中，想要找到 `continuedFromDll` 和 `fileSystem_api` 这两个异常，以及相关联的处理它们的信息。这清楚表明，如果一个异常延续的话，就可以得到关于该异常的更多的信息。

完整的例子程序可参见项目 `continueException\testContinuedException\testContinuedException.prj6`。在运行可执行程序 `testContunedException` 之前，必须首先建立项目 `continueException\continueException.prj6`。

本章小结

本章介绍了 Visual Prolog 6 的编码风格，内容包括基本元素、推荐格式、程序结构、程序设计语用学、存储管理，以及异常处理。这里描述的 Visual Prolog 程序的编码标准，是 Visual Prolog 系统本身的一部分。用户文档中的例子也是标准的，它们同样也代表了 Prolog 发展中心为用户推荐的编码标准。

习题 9

1. Visual Prolog 程序的编码标准指的是什么？Visual Prolog 程序的编码标准中，关键内容有哪些？
2. Visual Prolog 程序代码的格式指哪些方面？

3. Visual Prolog 程序设计语用学中，有哪些常规技巧？
4. 分析“截断”在 Visual Prolog 程序设计中的作用。
5. 指派输入格式的目的是什么？
6. 测试位于 `continueException\testContinuedException` 目录下的完整的例子程序 `test-ContinuedException.prj6`。

第 3 部分 语言参考

第 10 章 Visual Prolog 语言元素

第 11 章 Visual Prolog 数据元素

第 12 章 Visual Prolog 程序元素

第 13 章 编译单元

第 14 章 内部论域、谓词和常量

第 15 章 与其他编程语言接口

第 10 章 Visual Prolog 语言元素

本章介绍 Visual Prolog 6 程序设计语言的语法和语义。Visual Prolog 是基于逻辑程序设计语言 Prolog 的一种强类型的面向对象的程序设计语言。一个 Visual Prolog 程序包括一个目标、大量的接口声明和类的实现程序。

接口、类声明和类实现包括 Prolog 实体的定义和声明，即

- 论域
- 常量
- 谓词
- 事实数据库

Visual Prolog 程序的实际代码中的谓词定义由谓词声明和子句定义来声明。

10.1 类型

Visual Prolog 的类型 (type) 分为对象类型和数值类型。对象类型是可变的，数值类型是不可变的。

对象类型由接口 (interface) 进行定义。

数值类型包括数值型、字符串型、字符型以及复合论域。复合论域也可以看作是结构类型数据。复合论域的简化形式是结构和枚举类型，而更多的复杂形式是树型结构。

此外，Visual Prolog 有一种特殊类型，叫做引用论域，它可以由任意其他类型派生而来。引用类型与 Prolog 执行模型或语义密切相关，在下面将详细进行介绍。

类型以子类型层次结构进行组织。子类型用来引入包容多态性：希望某种类型的一个值能够同样接受任意的一个子类型值的任何上下文。或者倒过来说，在需要时把一定类型的值自动地转化为任意超类型，这样可以不需要显式的类型转换而访问该超类型。

子类型可源于除代数数据类型外的其他任意数值类型。源于代数数据类型的类型是同义类型而不是子类型，也就是说它们是同一类型而不是一子类型。

子类型的概念与子集概念密切相关。但是值得特别注意的是，尽管一个类型是另一类型子集的精确描述，但它并不需要成为一个子类型。一个类型只在特别声明时才是另一类型的子类型。例如：

```
domains
    t1 = [1..17].
    t2 = [5..13].
    t3 = t1 [5..13].
```

t1 是一个整型变量，取值从 1 到 17（包括端点在内）。同样，t2 取值从 5 到 13，但是

t2 不是 t1 的子类型。另外, t3 (包含与 t2 一样的取值) 则是 t1 的子类型, 因为这是声明了的。

语言中包含了少数隐含的子类关系, 但其他情况下子类关系都是在类型定义中具体规定的。

对象类型采用子类型层次结构组织, 该结构是源于预定义对象类型的对象的, 也就是说, 任意对象类型是一个对象的子类型。对象类型用接口相互支持的方式来规定。如果一个对象是支持某一其他接口的接口或对象类型, 那么该对象也具有那个类型并且能够不受限制地作为这样的对象应用。

10.2 对象系统

Visual Prolog 的对象系统 (object system) 包括外部视图 (external view) 和内部视图 (internal view)。

10.2.1 外部视图

本部分并不针对类进行介绍, 它旨在澄清 Visual Prolog 中类的概念。这里假设读者是熟悉普通类的概念的。这些描述不涉及任何语法及实现等内容, 而且也不考虑任何操作性或程序性的内容。同时, 介绍类、对象等的原因大都出于程序性的原因, 发现不涉及这些程序性原因来介绍基本概念是值得的。

Visual Prolog 的类的概念基于以下 3 项语义实体:

- 对象
- 接口
- 类

对象 (object)

一个对象是指若干命名对象成员谓词和一组支持接口的集合。实际上对象也有一个状态, 这个状态只能通过成员谓词来改变或观察, 称对象中的这种状态为封装。

接口 (interface)

一个接口是一种对象类型。它有一个名字且定义了一组命名对象谓词。

接口依据支持层次构建 (结构是接口对象底层的半网格结构)。如果一个对象有一个由接口指示的类型, 那么它也有这种任意支持接口的类型。因此, 这种支撑层次也是一种类型层。一个接口是它所支持的所有接口的子类。也就是说, 该对象支持该接口。如果一个接口名为 X, 那么就可以说该对象是一个 X, 或说一个 X 对象。

类 (class)

一个类是一个命名的对象工厂。它可以创建与某一接口相对应的对象。任何对象都由

类创建。如果类用接口 C 创建了一个对象，则称该对象为“对象 C”。

由一特定类创建的所有对象共享统一的对象成员谓词定义，但是每个对象又有其自身的状态。这样，对象成员谓词实际上是类的一部分，然而对象的状态却是对象自身的一部分。

一个类也包括另外一系列命名的谓词及一个封装的状态，分别作为类的成员和类的状态。类的成员和类的状态存在于每个基类中，然而对象成员和对象状态存在于每个基对象中。类的状态既可以通过类的成员访问，也可以通过对象成员访问。

注意：由一个类定义的一系列对象成员谓词是该类的接口中声明的谓词的联合。这特别意味着，如果不同的两个接口中声明了同一谓词，那么该类就只能为该谓词提供惟一的定义。因此，该类只有在含义清楚的时候才有效，也就是说，在这两个继承性的谓词的预定义语义相同的情况下，该类才合理。

此外接口支持必须清晰地指定。一些类提供与某些接口相应的谓词，这并不能说明该类就支持该接口。

模块 (module)

事实上，一个类根本不需要产生对象。这样的类可以只包含类的成员和类的状态。因此，这样的类与其认为是一个类，不如说是一个模块。

同一性 (identity)

每个对象都是惟一的：对象有可变的狀態，并且由于对象的状态可以通过它们的成员谓词进行观测，因此一个对象只与自身完全相同。也就是说，即使两个对象的状态完全相同，对象也是不同的，因为改变一个对象的状态并不能改变另一个对象的状态。

不能直接访问对象的状态，通常通过该对象的引用访问对象状态。同时，一个对象是与自身同一的。同一对象可以有许多的引用。这样，就可以通过许多不同的引用访问同一对象。

类和引用也是惟一的：它们通过自身的名字进行识别。两个引用或类在同一程序中不能有相同的名字（一个类和一个引用也不能名字相同）。

实质上，对象、类或接口的结构相同并不意味着它们就一样。

10.2.2 内部视图

10.2.1 小节从外部行为的角度介绍了对象、类和接口，本节将从内部问题的角度展开这些描述。

(1) 类的程序语义

这些内部问题更具有程序语义上的自然属性。可以考虑将类分为声明部分和实现部分。

从程序语义上的角度看，类是核心项，类中包含了代码。

接口主要具有静态价值。事实上，接口只存在于程序的正文描述中，并没有直接的运行描述。另一方面，对象则主要具有动态价值。在程序中对象并不能直接可见，它只有在程序真正运行时才存在。

一个类由声明和实现两部分组成。声明部分声明了类的公共存取部分以及产生的对

象。实现部分则定义了类声明中声明的实体。谓词的基本实现自然是子句，但是谓词也可以借助于继承进行定义，或由外部程序库解决。

Visual Prolog 中的类声明纯粹是陈述性的。它只声明可以存取的实体，而不声明怎样或在哪里实现实体。

类的实现可以声明和定义更多的实体（即论域、谓词等）。这些都只对于类自身是可见的，也就是说，它们是私有的。

一个对象的状态作为事实存储于该对象中。在类的实现中，这些事实作为正规的事实（数据库）段来声明。对于每个对象（正如其他的对象实体）而言，事实是局部的。类事实对于所有本类的对象都是共享的。

事实只能在一个类的实现中被声明。因此，不能被外部类直接访问。

类的实现也可以声明，它支持比声明部分所提及的更多的接口。但是，该信息只对于实现自身是可见的，因此是私有的。

（2）代码继承（code inheritance）

在 Visual Prolog 中，代码继承只能在类的实现中发生。Visual Prolog 支持多继承。可以通过在实现的一个特殊的继承部分中提及某一类以继承该类。继承类称为父类或超类。子类（child class）和支类（sub class）对父类而言是一样的，称该子类继承了父类。一个子类只能通过公用接口访问其父类，即在使用父类上，它并没有比其他类更多的特权。

10.3 作用域和可视性

本节介绍作用域和可视性（scoping & visibility），其中包括名字分类（name categories）、可视性、隐蔽及限定（visibility, shadowing & qualification）等。

10.3.1 名字分类

Visual Prolog 所有的名称（识别）主要可以分为两组：

- 常量名（以小写字母开头）
- 变量名（以大写字母或下划线开头）

常量名（标识符）分为以下几类：

- 类型名（即论域和接口）
- 论域载体（即类和接口）
- 不带圆括号的名称（即常量、事实变量和空变元算符）
- 变元 N 的有返回值的名字（即函数、算符和事实变量）
- 变元 N 的无返回值的名字（即谓词和事实）

Visual Prolog 要求在声明的地方不会出现名字冲突，因为在使用的地方无法解决冲突问题。只有在同一作用域内的声明才可能冲突，因为作用域限定可以用来解决冲突。一种分类内的名字永远不会与另一种分类内的名字相冲突，但是一个单独声明可能将一个名字置于不同的几个分类中。

(1) 一个接口和一个论域决不能在定义的地方产生冲突，因为所有接口都在全局范围内定义，同时所有的论域都定义在一个接口、类或实现内。两个接口不能同名，两个论域在同一作用域内不能以相同的名字声明。论域和接口之间的歧义可以用限定来解决（一个接口能够用 ‘::’ 限定，‘::’ 之前没有标识符）。既然预定义的论域（字符、字符串、符号、整数、无符号整数、实数、二进制、指针、布尔数、事实）也是全局的，那它们也可以用 ‘::’ 来限定，因此接口的名字绝不能与任何的论域名冲突。

(2) 类和接口不能同名，即两个类、两个接口或一个类和一个接口不能同名。除非当一个接口的命名以与该接口同名的类的构造类型使用时才允许同名。也就是可以使用下述语法：

```
interface interfaceAndClassName
    ...
end interface interfaceAndClassName
class interfaceAndClassName : interfaceAndClassName
    ...
end class interfaceAndClassName
```

因为类和接口只有一个作用域（即全局），所以不会有任何使用冲突。同名的类和接口能够声明域和常量。这些声明不能相互冲突，因为它们在同名空间中结束（因为只能用接口或与类相同的名字来限定）。

类系统对谓词、函数、常量和算符名称的使用有以下限制：定义域主要由限定和元数（arity）来决定，或与定义在当前作用域范围内的规则一起决定（除非使用了显式限定）。

也就是说：

- 如果以一个作用域名限定了名字，那么所定义的作用域当然由限定决定。
- 如果在当前作用域内定义了名字或元数，那么这就是定义域。
- 否则，作用域必须由名字和元数清楚地惟一决定（即名字或元数在两个或两个以上开放的或继承的作用域内被定义是错误的）。

定义规则仅仅取决于名字、元数和类型；必须有可能惟一地决定所定义的名字空间。

- 当前作用域覆盖所有其他作用域。必须使用限定（用 ‘::’ 限定）来解决名字空间冲突。
- 在单独的一个类型中，名字只能代表一个含义。

如果一个作用域声明一个不带括号的名称，那么它可以声明如下：

- 一个常量
- 一个事实变量
- 一个空变元算符

也就是说，不可能有一个与事实变量名或空变元算符同名的常量。

如果一个作用域声明一个有返回值的名称，那么它可以声明如下：

- 一个函数
- 一个算符
- 一个事实变量

也就是说，让一个函数、算符或事实变量同名、同变元是不可能的。

如果一个作用域声明一个无返回值的变量名字，那么可以声明如下：

- 一个谓词
- 一个事实

最后，关于重载。下列实体能够重载如下内容：

- 算符
- 函数
- 谓词

一个算符在同一论域内不可以使用两次（即使变元不同）。

10.3.2 可视性、隐蔽性及限定性

大多数的作用域规则都在上面提到过。这一节将使相关内容更为完整。

一个接口定义、类声明和类实现都有作用域（作用域不允许嵌套）。

一个实现（私有的）扩展了相应类声明的作用域。在一个作用域内的可视性都是相同的。这点特别是指在一个作用域内，不论在哪里声明的类型都在整个作用域内是可视的。

来自支持接口和超类的公用名，如果它们的来源没有歧义的话，那么在一个作用域内可以直接使用（即没有限制）。使用来源含糊的名字是非法的。在谓词调用中的所有歧义都会通过用类名对谓词名进行限定（比如，`cc::p`）来消除。

这一限制性也用于在当前对象中限定超类的对象成员谓词的调用。

Visual Prolog 有以下的隐蔽层次：

- 局部作用域
- 超类和开放作用域

开放作用域与超类地位相同，因此下面只介绍超类。

该层次是指一个局部声明会使一个超类声明隐蔽。但是，在超类之间并不隐蔽。所有的超类具有相同的优先权。如果两个或更多个超类包含冲突的声明，那么这些声明就只能通过限定进行访问。

举例

假设接口 `aa` 和类 `aa_class` 如下：

```
interface aa
    predicates
        p1 : () procedure ().
        p2 : () procedure ().
        p3 : () procedure ().
end interface
```

```
class aa_class : aa
end class
```

假设类 `bb_class` 如下：

```

class bb_class
  predicates
    p3 : () procedure ().
    p4 : () procedure ().
end class bb_class

```

在这些类的上下文中考虑类 `cc_class` 的实现:

```

implement cc_class inherits aa_class
  open bb_class
    predicates
      p2 : () procedure ().
      p5 : () procedure ().
    clauses
      new() :-
        p1(), % aa_class::p1
        p2(), % cc::p2 (shadows aa_class::p2)
        aa_class::p2(), % aa_class::p2
        p3(), % Illegal ambiguous call: aa_class::p3 or bb_class::p3
        aa_class::p3(), % aa_class::p3
        bb_class::p3(), % bb_class::p3
        p4(), % bb_class::p4
        p5(), % cc::p5
      end implement cc_class

```

10.4 词法结构

本节介绍 Visual Prolog 的词法结构 (lexical structure)，包括程序单元 (program elements)、标记 (tokens) 及文字 (literals)。

10.4.1 程序单元

Visual Prolog 编译器应用于源文件。该文件可以包括其他源文件，这些源文件插入最初的源文件构成一个编译单位。一个编译单位的编译依照两个步骤进行：

- (1) 输入被转换为标记序列；
- (2) 这些标记按语义分析并转化为可执行代码。

程序的词法分析将编译单元 (compilation Unit) 分割为一个输入元素的列表 (input Element)。

```

compilationUnit:
  inputElement-list

```

```
inputElement:  
    comment  
    whiteSpace  
    token
```

只有标记对于后续的语法分析才是重要的。

(1) 注释 (comments)

Visual Prolog 的注释以下列任意方式书写：

- /* (斜线, 星号) 字符, 后跟任意序列的字符 (包括新行), 以*/ (星号, 斜杠) 字符结束。这些注释可以是多行的, 也可以是嵌套的。
- % (百分号) 字符, 后跟任意序列的字符。以%字符开始的注释延续到行尾。因此, 它通常被称为“单行注释”。

观察下面的注释示例：

```
/* Begin of Comment1  
    % Nested Comment2 */ This mark does not close a multi-line comment because  
it is inside a single-line comment  
    This is the real termination of Comment1 */
```

(2) 空白 (whiteSpace)

```
whiteSpace:  
    blank  
    tab  
    newLine
```

这里的 blank 是一个空白字符, tab 是一个制表符, newLine 是一个新行。

10.4.2 标记

```
token:  
    identifier  
    keyword  
    punctuator  
    operator  
    literal
```

(1) 标识符 (identifiers)

```
identifier:  
    lowercaseIdentifier  
    uppercaseIdentifier  
    anonymousIdentifier  
    ellipsis
```

一个小写标识符 (lowercaseIdentifier) 是一个以小写字母开头的字母、数字和下划线的序列。一个大写标识符 (uppercaseIdentifier) 是一个以大写字母或下划线开头的字母、

数字和下划线的序列。

```
anonymousIdentifier : _
```

```
ellipsis :  
...
```

(2) 关键字 (keywords)

关键字分为主关键字和次关键字，这样的分类只是表面上的。主次关键字之间并没有真正的区别。在后文中以不同的颜色区分二者。

```
keyword :  
    majorKeyword  
    minorKeyword
```

```
majorKeyword : one of  
    class clauses constants constructors  
    div domains delegate  
    end  
    facts from  
    implement interface inherits  
    goal guards  
    mod monitor  
    open  
    predicates  
    resolve  
    supports
```

```
minorKeyword : one of  
    align and as anyflow  
    bitsize  
    determ digits  
    erroneous externally  
    failure  
    language  
    multi  
    nondeterm  
    or  
    procedure  
    reference  
    single  
    to
```

除了 `as` 和 `language` 外，所有的关键字都是保留字。

注意：`div` 和 `mod` 也是保留字，但是这些可归属于操作符类。在语言中不使用 `guards` 和 `monitor`，但是保留下来以备后用。

(3) 标点符号 (punctuation marks)

在 Visual Prolog 中, 标点符号对于编译器而言, 具有语法上的和语义上的 (syntactic and semantic) 含义, 但其本身并不指定可以产生值的运算符。某些标点符号或是单独的或是合成的, 都可以成为 Visual Prolog 的运算符。

标点符号如下:

```
punctuationMarks: one of
    ;    !    ,    .    #    [    ]    (    )    :-    :    ::
```

(4) 运算符 (operators)

运算符指定作用于操作数的一种运算。

```
operators: one of
    +    -    /    *    =    div    mod
    <    >    <>    ><    <=    >=    :=
```

所有的运算符都是二进制的, 运算符 “-” 和 “+” 是单目操作符。

div 和 mod 是保留字。

10.4.3 文字

程序文字分为以下几类: 整型、字符型、浮点型、字符串型、二进制及列表型。

```
literal:
    integerLiteral
    realLiteral
    characterLiteral
    stringLiteral
    binaryLiteral
    listLiteral
```

(1) 整数型文字

```
integerLiteral:
    octalPrefix octalDigit-list
    unaryMinus-opt decimalDigit-list
    hexadecimalPrefix hexadecimalDigit-list
unaryMinus:
    -
octalPrefix:
    0o
octalDigit: one of
    0 1 2 3 4 5 6 7
decimalDigit: one of
    0 1 2 3 4 5 6 7 8 9
hexadecimalPrefix:
    0x
```

```
hexadecimalDigit: one of
    0 1 2 3 4 5 6 7 8 9 A a B b C c D d E e F f
```

整型数不能超过最大、最小整数的范围。

(2) 实数型文字

```
realLiteral:
    unaryMinus-opt decimalDigit-list fractionOfFloat-opt exponent-opt
fractionOfFloat:
    . decimalDigit-list
exponent:
    exponentSymbol exponentSign-opt decimalDigit-list
exponentSymbol: one of
    e E
exponentSign: one of
    - +
```

浮点数不能超过最大、最小实数界限。

实数型文字需要小数部分和指数部分，否则被解释为整数。

(3) 字符型文字

```
characterLiteral:
    ' characterValue '
```

字符的值可以是任意可打印字符或一个转义序列：

\\ 代表 \
 \t 代表制表符
 \n 代表换行符
 \r 代表回车
 \' 代表单引号
 \" 代表双引号

一个“u”后面跟4位十六进制数代表相应数的双字节字符。

(4) 字符串型文字

```
stringLiteral:
    stringLiteralPart-list
```

```
stringLiteralPart:
    @" anyCharacter-list-opt "
    " characterValue-list-opt "
```

一个字符串由一个或多个 stringLiteralPart 串连接组成。如果 StringLiteralPart 以@开始，则不使用转义序列。因此，不使用@的字符串类型可以使用下述转义序列：

\\ 代表 \
 \t 代表制表符
 \n 代表换行符

\r 代表回车

\' 代表单引号（可直接表示为单引号字符）

\\" 代表双引号

一个“u”后面跟 4 位十六进制数代表相应数的双字节字符。

（5）二进制型文字

```
binaryLiteral:
    $[ elementValue-comma-sep-list-opt ]
elementValue:
    integerLiteral
```

elementValue 在[0..255]之间。

（6）列表型文字

列表型的元素可以是整型、字符型、浮点型、字符串型或二进制型。

```
listLiteral :
    [ simpleLiteral-comma-sep-list-opt ]
    [ simpleLiteral-comma-sep-list | listLiteral ]
```

这里，simpleLiteral 可以是如下内容：

```
simpleLiteral:
    integerLiteral
    realLiteral
    characterLiteral
    stringLiteral
    binaryLiteral
```

本章小结

本章介绍 Visual Prolog 6 程序设计语言的语法和语义，内容包括类型、对象系统、作用域和可视性，以及词法结构等。

Visual Prolog 是基于逻辑程序设计语言 Prolog 的一种强类型的面向对象的程序设计语言。一个 Visual Prolog 程序包括一个目标、大量的接口声明和类的实现程序。

习题 10

1. 认定一个系统是面向对象的，其准则有哪几点？什么叫对象的封装？
2. 接口与类有何关系？
3. 作用域指的是什么？
4. 可视性、隐蔽性及限定性的含义是什么？
5. Visual Prolog 的词法结构包括哪些内容？它们在实质上有何作用？

第 11 章 Visual Prolog 数据元素

本章介绍 Visual Prolog 的数据元素，内容包括论域段（domains sections）、通用类型和根类型（universal and root types）等。

11.1 论域段

一个论域段在当前作用域内定义一组论域（参见接口、类声明和类实现）。

```
domainsSection :  
    domains domainDefinition-dot-term-list-opt
```

（1）论域定义

一个论域定义，声明了一个当前作用域内已命名的论域。

```
domainDefinition :  
    domainName = typeExpression
```

如果右边的论域表示一个接口或一个复合论域，那么所定义的论域就是类型表达式的同义词（即完全相同）。否则，所定义的论域称为类型表达式所指示的论域的子类。这里，论域名 `domainName` 应当是小写标识符。

有些地方必须使用论域名，而不是类型表达式：

- 作为形式变元类型的声明；
- 作为一个常量或一个事实变量的类型；
- 作为列表论域中的类型。

（2）类型表达式

一个类型表达式指示一种类型。

```
typeExpression:  
    typeName  
    compoundDomain  
    listDomain  
    referenceDomain  
    predicateDomain  
    integralDomain  
    realDomain
```

11.1.1 类型名

一个类型名是一个接口名或者是一个值论域的名字。值论域这一术语用来指定元素不

可变的论域。也可以说，属于与接口名相一致的论域的对象具有可变的声明，任意其他论域的项都是不可变的。因此，实际上数值类型是除了对象类型以外的其他类型。一个类型名指示与现有论域的名称相应的类型。

```
typeName:
    interfaceName
    domainName
    classQualifiedDomainName
```

```
interfaceName :
    lowercaseIdentifier
```

```
domainName :
    lowercaseIdentifier
```

```
classQualifiedDomainName :
    className :: domainName
```

```
className :
    lowercaseIdentifier
```

这里，`interfaceName` 是一个接口名，`domainName` 是一个论域名，`className` 是一个类名。

举例：

```
domains
    newDomain1 = existingDomain.
    newDomain2 = myInterface.
```

其中，论域名是 `existingDomain`，接口名 `myInterface` 用来定义新的论域。

11.1.2 复合论域

复合论域（compound domains）也称作代数数据类型，用于表示列表、树和其他树形结构的数值。在其简单形式中，复合论域用于代表结构和枚举数值。复合论域可以递归定义，也可以相互或间接递归。

```
compoundDomain :
    alignment-opt functorAlternative-semicolon-sep-list
```

```
alignment :
    align integralConstantExpression
```

这里，`integralConstantExpression` 是一个表达式，它必须是在编译期间赋值为整数。

一个复合论域声明定义了一系列具有可选对齐方式（**alignment**）的算符选项。对齐方式项必须是 1, 2 或 4。

如果一个复合论域包含一个算符选项，那么它被视为结构，并且具有与 C 语言中的适当结构二进制兼容的表示法。

```
functorAlternative:
    functorName
    functorName ( formalArgument-comma-sep-list-opt )
```

这里，**functorName** 是一个算符选项的名称，它应当是小写标识符。**formalArgument** 的定义如下：

```
formalArgument :
    typeName argumentName-opt
```

这里，**argumentName** 可以是任意标识符，编译器忽略它。

复合论域是从它们派生出来的引用论域的子类，否则复合论域与任何其他论域都不具有这样的子类关系。

如果一个论域作为一个等价的复合论域进行定义，那么这两个论域是同义类型而不是子类型，即它们是同一类型的两个不同名字。

举例：

```
domains
    t1 = ff(); gg(integer,t1).
```

t1 是一个带有两个选项的复合论域。第 1 个选项是空变元算符 **ff**。第 2 个选项是两个变元的算符 **gg**，采用一个整型数论域和论域 **t1** 自身作为参数。因此，论域 **t1** 是递归定义的。

以下表达式是论域 **t1** 的项：

```
ff()
gg(77, ff())
gg(33, gg(44, gg(55, ff())))
```

举例：

```
domains
    t1 = ff(); gg(t2).
    t2 = hh(t1, t1).
```

t1 是一个带有两个选项的复合论域。第 1 个选项是空变元算符 **ff**。第 2 个选项是一元算符 **gg**，采用论域 **t2** 的项作为参数。**t2** 是一个带有一个选项算符 **hh** 的复合论域，算符 **hh** 采用两个 **t1** 项作为参数。因此，论域 **t1** 和 **t2** 是相互递归的。

以下表达式是论域 **t1** 的项：

```
ff ( )
```

```

gg (hh(ff(), ff()))
gg (hh(gg(hh(ff(), ff())), ff()))
gg (hh(ff(),gg(hh(ff(), ff()))))
gg (hh(gg(hh(ff(), ff())), gg(hh(ff(), ff()))))

```

举例:

在本例中, 论域 t1 与 t2 是同义类型。

domains

```

t1 = f(); g(integer).
t2 = t1.

```

通常, 在空变元算符后不需要带空括号。但是, 在一个仅由单个空变元算符构成的论域定义中, 要用空括号将其与一个同义字/子类定义区分开来。

举例:

t1 是一个仅有单个空变元算符的复合论域, 然而 t2 定义为 t1 的同义字。

domains

```

t1 = f().
t2 = t1.

```

11.1.3 列表论域

列表论域(list domains)代表一个特定论域的值序列。列表 T 中的所有元素必须是 T 类型。

```

listDomain :
    typeName *

```

T*是 T 元素列表的类型。

下列语法用于列表。

```

listExpression :
    [ term-comma-sep-list-opt ]
    [ term-comma-sep-list | constantName ]
    [ term-comma-sep-list | factVariableName ]
    [ term-comma-sep-list | functionCall ]
    [ term-comma-sep-list | variableName ]
    [ term-comma-sep-list | anonymousIdentifier ]
    [ term-comma-sep-list | listExpression ]

```

这里, functionCall 是一个函数调用, 返回一个 listDomain 类型的值。ConstantName, factVariableName 和 variableName 应当是 listDomain 类型。每项都应当是 typeName 类型。

实际上, 列表仅仅是带有两个算符的复合论域: []指示空列表; 算符[HD | TL]指示该列

表具有表头 HD 和表尾 TL。表头必须是基本元素类型，表尾必须是一个相关类型的列表。

因此，列表从语法上可以被修饰如下：

$[E1, E2, E3, \dots, En | L]$ 是 $[E1 \ [\ E2 \ [\dots \ [\ En \ | \ L \] \dots]]]$ 的简记。

$[E1, E2, E3, \dots, En]$ 是 $[E1, E2, E3, \dots, En \ []]$ 的简记，继而是 $[E1 \ [\ E2 \ [\dots \ [\ En \ | \ [] \] \dots]]]$ 的简记。

11.1.4 引用论域

一个引用论域（reference domains）与构造它的原始论域类似，除此之外，引用论域变量的值也可以是自由变量（即 `unknown`）。如果一个变量值是自由的，那么在合一过程中它可以被绑定到任意的论域值。变量一旦被绑定，通过回溯到被绑定值，该值被释放，否则一个已被绑定的变量是不可变的。如果该变量包含一个对象，那么该对象仍然具有可变状态。但是，该变量所绑定的对象不可改变。

```
referenceDomain :
    reference referenceDomainDescription
```

```
referenceDomainDescription :
    typeName
    compoundDomain
    listDomain
```

如果一个引用论域由一个复合论域构造，在算符中嵌套的所有论域也必须是引用论域。一个引用论域是一个基本的非引用论域的超论域。引用论域不是任何其他论域的子类型。

11.1.5 谓词论域

一个谓词论域（predicate domains）的值是具有相同“签名（signature）”的谓词。也就是说，具有相同的参数和返回类型，相同的流模式以及相同的（或加强的）谓词模式。

一个具有返回值的谓词称为函数，没有返回值的谓词被称为普通谓词，以强调它并不是一个函数。

```
predicateDomain :
    ( formalArgument-comma-sep-list-opt ) returnArgument-opt
    predicateModeAndFlow-list-opt callingConvention-opt
```

```
formalArgument :
    predicateArgumentType variableName-opt
    ellipsis
```

```
returnArgument :
    -> formalArgument
```



```
predicateArgumentType :
    typeName
    anonymousIdentifier
```

```
variableName :
    upperCaseIdentifier
```

谓词论域可以含有省略参数，就像作为列表 `formalArgument-comma-sep-list` 中的最后一个形参 `formalArgument` 一样。

谓词论域可以含有匿名标识符 `anonymousIdentifier` 来作为谓词参数类型 `predicateArgumentType`，以指定该参数可为任意类型。

目前，带有省略参数的谓词仅能用于谓词声明中。

用于论域定义中的谓词论域最多能声明一个流。

```
predicateModeAndFlow :
    predicateMode-opt flowPattern-list-opt
```

1. 谓词模式

当声明一个谓词时其模式可以省略。在一个实现内部（即对于一个局部谓词而言），所需的流和模式源自谓词的用法。在一个接口或一个类声明内部（即对于一个公有谓词而言），省略谓词模式意味着该谓词是一个过程 `procedure`，省略流模式意味着所有的参数都是输入参数。

为构造器声明一个谓词模式是非法的，这种谓词总是有 `procedure` 模式。

指定的谓词模式可应用于下列流模式的每个成员。

```
predicateMode : one of
    erroneous failure procedure determ multi nondeterm
```

谓词模式可用下列集合来描述：

```
erroneous = {}
failure = {Fail}
procedure = {Succeed}
determ = {Fail, Succeed}
multi = {Succeed, BacktrackPoint}
nondeterm = {Fail, Succeed, BacktrackPoint}
```

`Fail` 在集合中是指谓词失败。`Succeed` 在集合中是指谓词成功。`BacktrackPoint` 在集合中是指该谓词返回时会带一个活动的回溯点。

如果一个集合，比如说 `failure`，是另外一个集合的子集；比如说 `nondeterm`，那么可以说该模式是另外一个模式的加强模式，即 `failure` 是 `nondeterm` 的加强模式。

一个谓词论域实际上包含了所有具有指定模式或加强模式的谓词（带有正确的类型和流模式）。

2. 流模式

流模式（flow pattern）定义了参数的输入输出方向，这些参数与算符论域相结合，会成为带有单个输入参数的一些部分，以及相同输出参数的某些部分的结构。

一个流模式由一个流的序列组成，每个流对应一个参数（比如，第 1 个流对应第 1 个参数）。

```
flowPattern :
  ( flow-comma-sep-list-opt )
  anyFlow
```

```
flow :
  i
  o
  functorFlow
  listFlow
  ellipsis
```

省略流（ellipsis flow）模式必须与一个省略参数匹配，并且因此只能作为流模式中的最后一个流。

```
ellipsis :
  ...
```

一个算符流 `functorFlow` 声明了一个算符和构成该流的所有的流。当然，算符必须在相应参数的论域内。

```
functorFlow :
  functorName ( flow-comma-sep-list-opt )
```

一个算符流的声明不能包含省略流。

列表流恰恰与算符流类似，但却与列表论域具有相同的语法修饰。

```
listFlow :
  [ flow-comma-sep-list-opt listFlowTail-opt ]
```

```
listFlowTail :
  | flow
```

一个列表流不能包含省略流。

当声明一个谓词时，流模式可以被省略。在一个实现内部（即对于一个局部谓词而言），所需流模式源自该谓词的用法。在一个接口或一个类声明内部（即对于一个公有谓词而言），省略流是指所有参数均为输入参数。

特殊的流模式 `anyflow` 只能在局部谓词中进行声明（即在一个类的实现内部的谓词声明）。这意味着确切的流模式将在编译过程中估计出来。如果可选的先前谓词模式在这个流模式中被省略，那么就假定它是一个过程（`procedure`）。

举例:

```
domains
    pp1 = (integer Argument1).
```

pp1 是一个谓词论域。具有类型 pp1 的谓词带有一个整型变量作为参数。由于没有声明流模式，所以该参数被认为是输入参数；而且没有提到谓词模式，所以该谓词是过程 (procedure)。

举例:

```
domains
    pp2 = (integer Argument1) -> integer ReturnType.
```

类型 pp2 的谓词带有一个整型变量作为参数，并返回一个整型类型的值。因此，pp2 实际上是一个函数论域，具有类型 pp2 的谓词实际上是函数。由于没有声明流模式，所以该参数被认为是输入参数。而且没有提到谓词模式，因此该谓词是过程。

举例:

```
predicates
    ppp : (integer Argument1, integer Argument2) determ (o,i) (i,o) nondeterm (o,o).
```

谓词 ppp 有两个整型参数。它存在 3 个流模式变体：(o,i) 和 (i,o) 是确定性的；(o,o) 是不确定性的。

3. 调用约定

调用约定 (calling convention) 决定了参数等如何传递给谓词，也决定了从一个谓词名怎样获得一个连接名。

```
callingConvention :
    language callingConventionKind
```

```
callingConventionKind : one of
    c stdcall apicall prolog
```

如果没有声明一个调用约定，那么就假定用 Prolog 约定。Prolog 调用约定是一个用于 Prolog 谓词的标准约定。

调用约定 C 继承了 C/C++ 的标准调用约定。一个谓词的连接名用一个加前导下划线(_) 的谓词名创建。

调用约定 stdcall 采用 C 连接名策略，但是它用了不同的变量和堆栈处理规则。表 11.1 显示了 stdcall 调用约定的实现情况。

表 11.1 stdcall 调用约定的实现

特性	实现
参数传递顺序	从右至左
参数传递约定	按值传递，除非传递的是一个复合论域项或引用类型。因此不能用于参数数目变化的谓词

(续)

特性	实现
堆栈维护职责	被调用的谓词从堆栈中弹出自身参数
名称修饰约定	一个下划线(_)作为谓词名的前缀
大小写转换约定	不执行谓词名的大小写转换

调用约定 `apicall` 使用与 `stdcall` 相同的参数和堆栈处理规则，但是为了方便地调用 MS Windows API 函数，`apicall` 采用了大多数 MS Windows API 函数所用的名称约定。根据 `apicall` 的名称约定，一个谓词的连接名结构如下：

- 下划线符作为谓词名的前缀；
- 谓词名首字母用大写；
- 如果参数和返回类型指示 ANSI、双字节字符集和不确定性谓词，则分别用 A、W 作为后缀字符或无后缀；
- 以 @ 为后缀；
- 以压入调用堆栈的字节数为后缀。

举例

```
predicates
    predicateName : (integer, string) language apicall
```

该谓词的参数类型表明这是一个双字节字符集（正如 `string` 是双字节字符串论域一样）。在调用堆栈中，一个整型和一个字符串每一类型占 4 个字节，因此，连接名变成如下所示：

`_PredicateNameW@8`

如果 `apicall` 与 `as` 结构一起使用，则 `as` 结构中声明的名字以相同的方式被修饰。

`apicall` 只能在谓词声明中直接使用，而不能在谓词论域定义中直接使用。在谓词论域定义中取而代之的是 `stdcall`。

表 11.2 对比了 `c`，`apicall` 及 `stdcall` 调用约定的实现，而 `prolog` 调用约定实现比较特殊，这里不予以讨论。

表 11.2 c，apicall 及 stdcall 调用约定的实现对比

关键字	堆栈清理	谓词名大小写转换	连接谓词名修饰惯例
c	调用谓词从堆栈中弹出参数	无	下划线(_)作为谓词名前缀
stdcall	被调用的谓词从堆栈中弹出自身参数	无	下划线(_)作为谓词名前缀
apicall	被调用的谓词从堆栈中弹出自身参数	谓词首字母大写	下划线(_)作为名字前缀。首字母变为大写。A，W 作为后缀或无后缀。@符号作为后缀。参数列表中的字节数（十进制）作为后缀

Visual Prolog 谓词论域的概念涵盖了类和对象成员。类成员直接被处理，但是对象成员的处理却需要注意。一个对象谓词的调用将在成员所属对象的上下文中获得。

假定有下列声明：

```
interface actionEventSource
    domains
        actionListener = (actionEventSource Source) procedure (i).
    predicates
        addActionListener : (actionListener Listener) procedure (i).
    ...
end interface
```

同样假定类 `button_class` 支持 `actionEventSource`。当单击按钮时事件被发送。在 `myDialog_class` 类中，实现一个对话框，这是为了能够响应单击按钮操作而创建了一个按钮，以便监听活动事件。

```
implement myDialog_class
    clauses
        new() :-
            OkButton = button_class::new(...),
            OkButton::addActionListener(onOk),
            ...
    facts
        okPressed : () determ.
    predicates
        onOk : actionListener.
    clauses
        onOk(Source) :-
            assert(okPressed()).
end implement
```

值得注意的是，`onOk` 是一个对象成员。当单击按钮时，已注册的 `onOk` 的调用将在拥有 `onOk` 的对象中返回。也就是说，有权访问对象事实 `okPressed`。

11.1.6 整型论域

整型论域用于表示整型数，主要分为符号整数和无符号整数两类。整型论域也可以有不同的表示范围。预定义论域 `integer` 和 `unsigned` 代表有符号和无符号整数，表示的长度与处理器体系结构的自然长度相同（即 32 位机器就是 32 位）。

```
integralDomain :
    domainName-opt integralDomainProperties
```

如果在 `integralDomainProperties` 之前声明 `domainName`，那么这个论域本身必须是整型论域，其派生的论域也必须是这一论域的子类型（`child type` 或 `sub type`）。这种情况下，`integralDomainProperties` 不可以违背作为子类型的可能性，也就是说，其范围不可扩展，

大小不可改变。

```
integralDomainProperties :
  integralSizeDescription integralRangeDescription-opt
  integralRangeDescription integralSizeDescription-opt
```

```
integralSizeDescription :
  bitsize domainSize
domainSize :
  integralConstantExpression
```

整型大小描述声明了整型论域以位计算的大小 `domainSize`。编译器实现整型论域的这种表示，它不少于所指定的位数。`domainSize` 的值应当是确定的，并且不超过编译器所支持的最大值（在版本 6.x 中是 32 位）。

如果整型范围描述被省略，那么就与父论域相同。如果没有父论域，那就记为处理器的自然长度。

```
integralRangeDescription :
  [ minimalBoundary-opt .. maximalBoundary-opt ]
minimalBoundary :
  integralConstantExpression
maximalBoundary :
  integralConstantExpression
```

整型范围描述声明了限定整型论域的最小界（`minimalBoundary`）和最大界（`maximalBoundary`）。如果该限定被省略，那么就用父论域的限定范围。如果没有父论域，那么用 `domainSize` 分别决定这个最大值和最小值。

注意，指定的最小值不能超过指定的最大值，即

```
minimalBoundary <= maximalBoundary
```

最小界和最大界必须满足 `bitsize` 所隐含规定的位长度。

论域位长度（`domainSize`）的值，以及最小界和最大界的值必须在编译期间进行计算。

11.1.7 实型论域

实型论域用于表示带有小数部分的数（即浮点数）。实型论域可用于表示非常大和非常小的数。内部论域 `real` 具有处理器体系结构的自然精度（也是编译器所赋予的精度）。

```
realDomain :
  domainName-opt realDomainProperties
```

如果在 `realDomainProperties` 之前声明 `domainName`，那么这个论域本身就必须是一个实数论域，并且结果论域是这个论域的子类型。在此情况下，`realDomainProperties` 不能违

背作为一个子类型的可能性，也就是说，范围不能被扩展，精度不能被增加。

```
realDomainProperties :
    realPrecisionDescription realRangeDescription-opt
    realRangeDescription realPrecisionDescription
```

```
realPrecisionDescription :
    digits integralConstantExpression
```

实型精度描述声明了实型论域的精度，精度由小数位数决定。如果精度省略，则与父论域相同。若没有父论域，那么就取处理器的自然精度或是编译器所指定的精度（在 Visual prolog 6 中，编译器限制为 19 位数）。编译器会给精度一个上限和一个下限，如果精度超过了所用限度，就只获得处理器（编译器）所指定的精度。

```
realRangeDescription :
    [ minimalRealBoundary-opt .. maximalRealBoundary-opt ]
minimalRealBoundary :
    realConstantExpression
maximalRealBoundary :
    realConstantExpression
```

这里，`realConstantExpression` 是一个表达式，必须在编译时间内计算出浮点数值。即在编译时间内，实型论域的精度和限制都必须计算出来。

实型论域描述声明了实型论域的上下限。如果这一限制被省略，那就是与父论域相同。如果没有父论域，那么就将使用最大可能的精度范围。

注意，指定的最小界不能超过指定的最大界，即

```
minimalRealBoundary <= maximalRealBoundary
```

而且，最小界和最大界应当满足指定精度数位所隐含规定的限制范围。

11.2 通用类型和根类型

Visual Prolog 使用某些称为根类型（root types）和通用类型（universal types）的整数类型。

11.2.1 通用类型

一个数字文字比如 1，不具有任何特殊的类型，它可以用作包括实型在内的任意包含 1 的类型的值。

可以说，1 具有一个通用类型。通用类型指能够表示其值的任意类型。算术运算也返回通用类型。

11.2.2 根类型

算术运算对操作数的运算限制极低：任意整型论域的整数都可以相加。

也就是说，算术操作数是将根类型作为参数的。整数根类型是任意整型的超类型（即使声明中并没有提及）。因此，任意整型都可以转化为整数根类型。并且，由于算术运算因根类型而存在，就意味着它们其中之一将作用于任意整型论域。

关于根类型的具体数目以及操作数的存在形式与基本通用软件库设施有关，不在讨论范围之内。

本章小结

本章介绍 Visual Prolog 的数据元素，内容包括论域段（domains sections）、通用类型和根类型（universal and root types）等。论域段有简单论域、复合论域、列表论域、引用论域、谓词论域、整型论域及实型论域等。

习题 11

1. Visual Prolog 的论域段用来声明谓词参数的非标准论域。它与其他语言中的类型一样吗？
2. 这里的谓词与其他语言的函数和过程有关系吗？
3. 引用论域与其他论域有何本质区别？它有什么作用？
4. 引入通用类型和根类型有什么意义？

第 12 章 Visual Prolog 程序元素

本章介绍 Visual Prolog 的程序元素，内容包括项（terms）、常量、谓词、子句、事实、运算、程序段等。

12.1 项

本节介绍项的有关概念，内容包括运算符、类成员的访问、对象成员的访问全局实体的访问，以及论域、算符和常量的访问等。

12.1.1 项的基本概念

项有两种类型：公式（formulas）和表达式（expressions）。

- 表达式代表数值，比如数字 7；
- 公式代表逻辑声明，比如“数字 7 比数字 3 大”。

下面是项的简化定义，其中包括非法的语法结构。例如，! $+$!的书写形式是不合法的。但是，相信和语言概念的直觉理解相结合时，使用这样的简化语法表示，在大多数情况下对于类型系统和运算符层次结构的描述更为有利。

```
term:
  ( term )
  unaryOperator term
  term binaryOperator term
  literal
  identifier
  qualifiedName
  globalName
  memberAccess
  predicateCall
  cut
  ellipsis
  factvariableAssignment
```

(1) 文字有通用类型

```
literal:
  stringLiteral
  characterLiteral
  integerLiteral
```

```
realLiteral
binaryLiteral
```

```
cut:
    !
```

(2) 谓词调用

一个谓词调用形式如下:

```
predicateCall:
    term ( term-comma-sep-list-opt )
```

首项必须直接声明调用谓词的名字。也就是说, 首项必须是一个谓词名, 一个限定谓词名或一个成员访问。

注意: 有些谓词有返回值, 有些则没有返回值。如果一个谓词有返回值, 这个值就必须在上下文中使用, 不能被忽略。

(3) 事实赋值

赋值操作符 `:=` 用于向事实变量 (`factVariable`) 赋予一个新值。项必须被赋予一个适当类型 (即与事实变量或子类型相同的类型) 的值。

```
factVariableAssignment:
    factVariable := term
```

12.1.2 运算符

运算符 (`operators`) 按优先层次进行组织。在规则中, 下面各组中的操作符具有相同的优先权, 并且上面的操作符比下面的优先级高。也就是说, 一元加减法要比乘法运算符优先级高, 而乘法运算符又比加法运算符高。圆括号可以改变运算优先级。

```
unaryOperator:
    - +
```

```
binaryOperator:
    multiplicationOperator
    additionOperator
    relationOperator
    andOperator
    orOperator
```

1. 算术运算符

算术运算符 (`arithmetic operators`) 用于数字的算术运算。它们是表达式, 用表达式作为参数。它们采用根类型作为参数, 并返回通用类型的结果 (参见通用类型和根类型)。所

有的算术操作符都是左结合的（left associative）。

```
multiplicationOperator: one of
* / div mod
```

```
additionOperator: one of
+ -
```

2. 关系运算符

关系运算符（relational operators）是公式，用表达式作为参数。它们本质上是无关联的。

```
relationOperator: one of
> < > = <= <> >< =
```

3. 逻辑运算符

逻辑运算符（logical operators）主要包括逻辑“与（and）”、逻辑“或（or）”及逻辑“非（not）”运算符等。逻辑“与”运算符（andOperator）和逻辑“或”运算符（orOperator）是公式，用公式作为参数。它们是左结合的。“,”和 and 是同义词，“;”和 or 也是同义词。

```
andOperator: one of
, and
```

```
orOperator: one of
; or
```

举例：

以下的项

```
7 + 3 * 5 * 13 + 4 + 3 = X / 6 ; A < 7, p(X)
```

与下面的项含义相同：

```
((((7 + ((3 * 5) * 13)) + 4) + 3) = (X / 6)) ; ((A < 7) , p(X))
```

也就是说，项的最外面一层是两个项的 or。第 1 项是一个(=)关系项，第 2 项是 and 关系项。

12.1.3 类成员访问

类实体通过限定类名的方式进行访问：

```
qualifiedName:  
    identifier :: identifier
```

这样的限定名像普通的名字一样使用，即如果它是一个谓词，它就可以用于一个参数集。有些名字访问不需要限定，参见有关作用域的内容。

12.1.4 对象成员访问

每当引用一个对象时，都可以访问该对象的对象成员谓词。

```
memberAccess:  
    term : identifier
```

目前，项必须是一个变量或一个事实变量。

标识符（`identifier`）必须是项的类型。

在一个实现内部，对象成员谓词不需要引用对象就可以被调用，因为 `This` 已经被包含在其中了。参见有关作用域的内容。

12.1.5 全局实体的访问

存在于 `Visual Prolog` 中的仅有的全局实体是类、接口和内部论域、谓词、常量。全局名在任意作用域内都可以直接访问。也可能存在全局名与局域名或输入名重合的情况。在这种情况下，全局实体可以用双冒号 ‘`::`’ 来限定（不带前缀的类名或接口名）。双冒号可以随处使用，但是最重要的用处还是接口名用作形式参数类型说明符的情况。

```
globalName:  
    :: identifier
```

12.1.6 论域、算符和常量访问

论域、算符和常量都像类成员一样被访问。即使它们在一个接口中被声明，即当它们要被限定的时候，也总是以类或接口名加双冒号来限定。

12.2 常量

本节介绍常量（`constant`）的有关概念，内容包括常量段、常量定义等。

12.2.1 常量段

一个常量段（`constants section`）定义了当前作用域内的常量集。

```
constantsSection :  
    constants constantDefinition-dot-term-list-opt
```

12.2.2 常量定义

常量定义 (constant definitions) 声明一个命名的常量, 包括它的类型和值。

```
constantDefinition :  
    constantName : typeName = constantValue
```

```
constantName :  
    lowerCaseIdentifier
```

常量值 (constantValue) 是一个表达式, 在编译时间内计算。常量名 (ConstantName) 应该是一个小写标识符 (lowerCaseIdentifier)。

如果一个类型名 (typeName) 论域是一个标准论域, 那么它和冒号 ‘:’ 可以被省略, 得到以下简写形式:

```
constantDefinition :  
    constantName = constantValue
```

以这样的方式定义的常量可以用于所有的上下文中, 在这里可以使用与其同一种的文字。

如果类型名被省略, 那么常量论域必须明确地被常量值表达式确定。仅在下列内部论域情况下, 类型名才能被省略:

- (1) 数字 (整数或实数) 常量。在这种情况下, 相应的匿名数字论域被采纳为常量 (详细情况参见数字论域)。
- (2) 二进制常量。
- (3) 字符串常量。
- (4) 字符常量。

12.3 谓词

本节介绍谓词 (predicates) 的有关概念, 内容包括谓词段 (predicates sections)、构造段 (constructors sections)、接口谓词 (predicates from interface)、谓词的元 (arity) 等。

12.3.1 谓词段

谓词段声明当前作用域内的对象或类谓词的集合。

```
predicatesSection :
```

```
class-opt predicates predicateDeclaration-dot-term-list-opt
```

关键字 `class` 只能在类实现内部使用，这是因为在接口中声明的谓词永远是对象谓词；在类声明中声明的谓词永远是类谓词。

类声明用于声明作用域中的谓词，在作用域中这些谓词是可见的。当谓词在一个接口定义中被声明时，则相应类型的对象必须支持这些谓词。当谓词在类声明中被定义时，则该类提供所声明的公用谓词。并且，如果谓词在类的实现中被声明的话，那么该谓词就在局部可用。在所有的情况下，必须存在谓词的相应定义。

```
predicateDeclaration :  
    predicateName : predicateDomain linkName-opt  
    predicateName : _predicateDomainName linkName-opt
```

```
linkName :  
    as stringLiteral
```

```
predicateName :  
    lowerCaseIdentifier
```

这里，`predicateDomainName` 是在论域段声明的谓词论域名。

一个谓词声明，声明了该谓词的名称以及类型、模式、流（参见谓词论域）和可选的一个连接名。

只有类谓词可以有连接名。如果没有声明连接名，那么就从谓词名取连接名，取名的方式取决于调用约定。

如果调用约定是 `apicall`，那么 `as` 子句中所声明的连接名就可以采用任意方式修饰；如果该修饰不是所要的，则用 `stdcall` 代替。

12.3.2 构造段

构造段声明了构造器的集合。这些构造器属于所出现的构造段的作用域（参见类声明和类实现）。

```
constructorsSection :  
    constructors constructorDeclaration-dot-term-list-opt
```

构造段只能出现在构造对象的类的声明和实现当中。

构造声明用于声明类的已命名的构造器。

实际上，构造器包含两个相关谓词。

- 一个类函数，返回一个新的被构建的对象；
- 一个对象谓词，在初始化继承对象时使用。

一个相关的对象谓词构造器用于执行一个对象的初始化。该谓词只能从该类自身的构造器或是该类的继承类的构造器中调用（即基本的类初始化）。

```

constructorDeclaration :
    constructorName : predicateDomain

```

给构造器声明谓词模式是非法的，构造器总是过程模式的。

考虑下面的类：

```

class test_class : test
    constructors
        new : (integer Argument) .
end class test_class

```

相关的类级的谓词形式如下（有以下标志）：

```

class predicates
    new : (integer) -> test.

```

相关的对象级的谓词形式如下：

```

predicates
    new : (integer).

```

再考虑下面的实现：

```

implement test2_class inherits test_class
    clauses
        new() :-
            test_class::new(7), % invoke the base class constructor on
            "This"
            p(test_class::new(8)). % create a new object of the base class
            and pass it to p(...)
        ...

```

第 1 次调用 `test_class::new` 不返回值，因此这是对构造器的非函数对象的一次调用。也就是说，这是基本类构造器 `This` 的一次调用。

第 2 次调用返回一个值，因此它是对类的构造器的函数调用。也就是说，创建了一个新的对象。

12.3.3 接口谓词

一个接口能够通过 `predicates from` 段声明的谓词来支持另一接口的子集。`predicates from` 段指定接口和所有支持的谓词。这些谓词以名字或者名字和元数来声明。

如果一个接口支持另一个接口的子集，那么它就不是与另外那个接口相关的子类型或超类型。

关于 `predicates from` 段重要的是，所提及的谓词保留它们的原始接口。因此：

- 来自原始接口的任意谓词不会发生支持冲突；
- 它们能够作为来自原始接口的谓词被继承。

```

predicatesFromInterface :
    predicatesfrominterfaceNamepredicateNameWithArity-comma-sep-list-opt

```

predicatesFromInterface 只能在接口定义中被使用。

举例：

```

interface aaa
    predicates
        ppp : ().
        qqq : ().
end interface aaa
interface bbb
    predicates from aaa
        ppp
    predicates
        rrr : ().
end interface bbb
interface ccc supports aaa, bbb
end interface ccc

```

即使 aaa 和 bbb 都声明了谓词 ppp, 但 ccc 可以不产生任何冲突地支持二者。这是因为, ppp 在所有情况下都含有 aaa, 将其作为原始接口。

举例：

```

interface aaa
    predicates
        ppp : ().
        qqq : ().
end interface aaa
interface bbb
    predicates from aaa
        ppp
    predicates
        rrr : ().
end interface bbb
class aaa_class : aaa
end class aaa_class
class bbb_class : bbb
end class bbb_class
implement aaa_class inherits bbb_class
    clauses
        qqq().
end implement aaa_class

```

aaa_class 可以从 bbb_class 继承 ppp, 因为 ppp 在两个类中都含有 aaa, 并将其作为原

始接口。

12.3.4 变元

使用 N 个参数的谓词被称为 N 元谓词 (N -ary)，或者说该谓词有 N 个变元。所含变元数不同的谓词，即使它们名称相同，通常也是不同的谓词。

在大多数情况下，一个谓词的元数在包含该谓词的上下文中是明显的。但是，在某些情况，比如，`predicatesFromInterface` 段和 `resolve` 限定中，变元数并不明显。

为了区别 `predicates from` 段和 `resolve` 限定中不同变元数的谓词，谓词名可以选择采用带有变元数的声明。

下列变元数是可能的：

- `Name/N` 指一个普通谓词（即不是一个函数）的名字，变元个数为 N ；
- `Name//N` 指一个函数名，变元个数为 N ；
- `Name/N...` 指一个带 N 个变元的普通谓词名，后跟一个省略参数（即个数可改变的参数）；
- `Name//N...` 指一个带 N 个变元的函数名，后跟一个省略参数。

```
predicateNameWithArity :
    predicateName arity-opt
```

```
arity :
    / integerLiteral ellipsis-opt
    // integerLiteral ellipsis-opt
```

在 `Name/0...` 和 `Name//0...` 中，0 是可选项，因此它们可以分别写作 `Name/...` 和 `Name//...`。

注意：省略号 (...) 可以作为最后一个形式参数用于谓词和谓词论域的声明。在这种情况下，是指所声明的谓词（或谓词论域）的参数个数是可改变的。省略号必须与一个省略参数匹配，因此只能是流模式中的最后一个流。

12.4 子句

本节介绍子句 (clauses) 的有关内容，包括子句段 (clauses sections)、目标段 (goal sections) 等。

12.4.1 子句段

子句段由子句集组成。子句段包括谓词的实现或事实的初始化值。

一个单独的子句段可以含有几个谓词和事实的子句。另一方面，同一谓词或事实（同

名并变元数相同)的所有子句必须集中在一个子句段中,并且不涉及其他谓词或事实的子句。

```
clausesSection :
    clauses clause-dot-term-list-opt
```

子句用于定义谓词。单一的谓词由一个子句集定义。每个子句依次执行,直到它们执行成功,或没有子句可执行为止。如果没有子句成功,该谓词失败。

如果一个子句成功,并且在一个谓词的剩余部分有更多相关子句,那么程序控制将在以后回溯到该谓词子句,以查找其他的解决方案。

这样,一个谓词可以失败、成功,甚至成功多次。

每个子句有一个子句头(head)和一个可选的子句体(body)。

当调用一个谓词时,子句从顶部到底部依次试验。对于每个子句,子句头由来自调用的参数进行合一。如果这样的合一成功,则子句体(如果存在的话)就被执行。如果子句头匹配成功且子句体执行成功,则该子句成功。否则,子句失败。

有关内容也可参见评估部分。

子句由一个子句头(head)和一个可选的子句体(body)组成。

```
.
clause :
    clauseHead returnValue-opt clauseBody-opt
```

```
clauseHead :
    lowercaseIdentifier ( term-comma-sep-list-opt )
```

```
returnValue :
    = term
```

```
clauseBody :
    :- term
```

12.4.2 目标段

目标段是一个程序的入口。当程序开始执行时,首先从目标段开始执行,目标段被执行完后,程序就退出。

```
goalSection :
    goal term .
```

目标段由一个子句体组成。目标段定义了它自身的作用域,因此所有的调用都应当包含类的限定符。

通常,目标必须是过程模式。

12.5 事实

本节介绍事实 (facts) 的有关内容, 包括事实段 (facts sections)、事实声明 (fact declarations)、事实变量 (fact variables) 等。

12.5.1 事实段

一个事实段声明一个由若干事实组成的事实数据库。该事实数据库及事实属于当前作用域。

事实数据库可以存在于类级别上, 也可以存在于对象级别上。

事实段只能在类实现中进行声明。

如果对该事实数据库命名, 那么就隐含地定义了一个附加的复合论域。这个复合论域与事实段同名, 并且含有与这个事实段中的事实相应的算符。

如果对事实数据库命名, 那么这个名字就是指内部论域 factDB 的值。谓词 save 和 consult 接受这个论域的值。

```
factsSection :
    class-opt facts factsSectionName-opt factDeclaration-dot-term-list-opt
```

```
factsSectionName :
    - lowerCaseIdentifier
```

12.5.2 事实声明

事实声明用于声明一个事实数据库的事实。事实声明也是一个事实变量或一个事实算符。

```
factDeclaration :
    factVariableDeclaration
    factFunctorDeclaration
```

```
factFunctorDeclaration :
    factName : ( argument-comma-sep-list-opt ) factMode-opt
```

```
factName :
    lowerCaseIdentifier
```

一个事实算符声明默认为 nondeterm 事实模式。

一个事实算符可以通过子句段进行初始化。在这种情况下, 子句中的值应当是表达式, 这些表达式可以在编译时间内进行求值。

```
factMode : one of
    determ nondeterm single
```

如果模式为 `single`，那么一个事实就有一个值并且只有一个值，而且谓词 `assert` 会给原来的值赋新的值。谓词 `retract` 不用于单个事实。

如果模式为 `nondeterm`，那么这一事实就可以有 0, 1 或任意其他个值。如果模式为 `determ`，那么事实可以有 0 或 1 个值。如果事实有 0 个值，那么任何读操作都会失败。

12.5.3 事实变量

一个事实变量与一个一元单个事实算符类似。但是，从语法上讲，它作为可变变量（即以赋值方式）使用。

```
factVariableDeclaration :
    factVariableName : domain initialValue-opt
```

```
initialValue :
    := constantValue
```

```
factVariableName :
    lowerCaseIdentifier
```

一个常量值（`constantValue`）应当是一个项（论域类型的），可以在编译时间求值。

只要在一个构造器中将事实变量初始化，那么常量值就可以省略。类事实变量应当总有一个初始的常量值。

注意：关键字 `erroneous` 可被用来将其值赋给事实变量。

下面两行是有效的：

```
facts
    thisWin : vpiDomains::windowHandle := erroneous.
clauses
    p() :- thisWin := erroneous.
```

用 `erroneous` 赋值的基本思想，是为了在某些代码错误地使用了未初始化的事实时，给出一个明确的运行时间错误。

12.5.4 事实

事实只能在类实现中进行声明，并且以后它们只能从这个实现被引用，因此事实的作用域就是它们被声明的这个实现。但是，对象事实的生命期是它所属的对象的生命期。同样，类事实的生命期是从程序开始到程序结束。

下面的类声明了一个对象事实（objectFact）和一个类事实（classFact）：

```
implement aaa_class
  facts
    objectFact : (integer Value) determ.
  class facts
    classFact : (integer Value) determ.
    ...
end implement aaa_class
```

12.6 评估

评估（evaluation）又称为求值运算。Visual Prolog 通过执行目标来实现。目标是一个项。本节叙述项和子句的执行或计算是怎样进行的，包括回溯（backtracking）、谓词调用、合一（unification）、引用论域、匹配、嵌套的函数调用、变量与常量、算术表达式、事实断言与撤销等。

12.6.1 回溯

一个 Prolog 程序的评估或运算是搜索求解的过程。搜索求解的每步或者成功或者失败。在程序执行的特定点上，有可能不止有一种解决方案。当遇到这样的选择点时，就建立所谓的回溯点。一个回溯点是程序状态的一个记录，即添加一个指针到未执行的选择点。如果它证明了初始的选择不能提供解决方案（即失败），那么程序将回溯到记录过的回溯点，从而恢复程序状态和追踪另一个选择。在下面的部分还将对该机制进行详细描述和解释。

12.6.2 谓词调用

通过使用参数到一个谓词实现该谓词的调用。该谓词必须有一个流模式，以匹配参数的自由或绑定状态。每个谓词由一个子句集合（或在外部用某种其他语言）定义。

当谓词被谓词调用引用时，每个子句依次执行，直到它们成功，或直到没有子句可执行为止。如果没有子句成功，那么该谓词失败。

如果一个子句成功，并且还剩有其他的有关子句，那么程序控制可以在以后回溯到剩余的子句，以搜索其他的解。

这样，一个谓词可能失败、成功，甚至成功多次。

每个子句有一个子句头和一个可选的子句体。

当一个谓词被调用时，子句依次从顶部到底部被尝试。对于每个子句，在子句头的变量与来自调用的参数合一。如果合一成功，那么执行子句的子句体（如果它存在的话）。如果与子句头匹配且子句体执行成功，则该子句成功。否则，该子句失败。

举例:

```
clauses
  ppp() :- qqg(X), write(X), fail.
  qqg(1).
  qqg(2).
  qqg(3).
```

当 ppp 被调用时, 它依次调用 qqg。当 qqg 被调用, 它首先建立一个指向第 2 个子句的回溯点。然后执行第 1 个子句。因此 ppp 中的自由变量 X 与数字 1 匹配, 从而 X 被绑定为 1。

在 ppp 执行时, X (即 1) 被写出后, 由 fail 援引回溯点, 从而程序控制被设置到 qqg 中的第 2 个子句, 并且程序状态被设置回 qqg 首次进入的状态, 即 ppp 的 X 不再被绑定。

在 qqg 中的第 2 个子句实际执行之前, 第 3 个子句的回溯点已经建立好了。

12.6.3 合一

当一谓词被调用时, 来自调用的参数与每个子句的子句头的项合一。

合一是绑定变量的过程, 在这一过程中, 两项通过尽可能少的绑定达到相等 (即为进一步绑定留下尽可能多的开放空间)。

变量可以绑定为各种项, 包括变量或含有变量的项。

合一有时可能有时不可能, 也就是说它可能成功也可能失败。

变量和与其合一的项是有类型的, 一个变量只能被绑定到与它类型相同的项或子类型上。当两个变量互相绑定时, 它们就必须是完全相同的类型。

如上所述, 合一发生在一个谓词调用和子句头之间, 也发生在比较两项是否相等之时。考虑类型相同的两项:

```
T1 = f1(g(), X, 17, Y, 17)
T2 = f1(Z, Z, V, U, 43)
```

打算从左至右合一这两项 (即从左至右前向遍历)。

T1 和 T2 都是 f1/5 项, 这是匹配。因此打算将 T1 中的参数与 T2 中相应的参数进行合一。首先必须合一 Z 和 g(), 如果将 Z 绑定到 g(), 这个合一就可以实现。目前, 一切正常, 产生了合一器中的第 1 个绑定:

```
Z = g()
```

接下来的两个参数是 X 和 Z, 其中 Z 已经被绑定到 g()。如果将 X 也绑定到 g(), 那么这两个参数也可以合一。因此现在对于合一器有如下的新增内容:

```
X = Z = g()
```

再下来必须将 **V** 绑定为 17，然后将 **Y** 绑定到 **U**。再合并到合一器中得

```
X = Z = g()  
V = 17  
Y = U
```

现在，这两个合一的项等同于下面的项：

```
T1 = f1(g(), g(), 17, Y, 17)  
T2 = f1(g(), g(), 17, Y, 43)
```

但是最后两个参数 17 和 43 不能进行合一，没有变量绑定可以使这两项相等，所以总的来说，合一失败。

T1 和 **T2** 不能合一。

在上面的例子中，**T1** 原本应是一个谓词调用，**T2** 是子句的子句头，但是它们也成为了等式比较的两项。

12.6.4 引用论域

下面举例说明引用论域的概念和用法。

考虑类型相同的两项：

```
T1 = f1(g(), X, 17, Y)  
T2 = f1(Z, Z, V, U)
```

这些项可以依照以下合一器进行合一：

```
X = Z = g()  
V = 17  
Y = U
```

并且合一后的项如下：

```
f1(g(), g(), 17, Y)
```

这一项包括一个自由变量 **Y**（它正好绑定到 **U**）。如果项的类型为引用论域，那么它们只能是自由变量。

12.6.5 匹配

对于不是引用论域类型的项不需要进行完全彻底的合一，用匹配就足够了。如果该项包括引用论域类型的子项，那么这些子项必须合一而不是匹配。

除了变量只能被绑定到基础（grounded）项之外，匹配与合一是相同的。一个基础项是不包含任何自由变量的项。

为谓词声明适当的流模式，就有可能使用匹配而不是完全彻底地进行合一。

举例:

```
clauses
    ppp(Z, Z, 17).
    qqg() :-
        ppp(g(), X, 17).
```

ppp 调用与 ppp 子句的合一可能使用下面的合一器:

```
Z = X = g()
```

如果 ppp 有流模式(i,o,i), 那么合一就是一个匹配:

- g()作为第 1 个参数输入, 绑定到 Z。
- 子句中的第 2 个参数因此被绑定; 这样输出 X, X 被绑定到 g()。
- 最后第 3 个参数 17 作为输入, 这个数字只是与子句中的第 3 个参数进行比较。
- 只有通过流模式才能预测子句不需要真正合一的可能性。

12.6.6 嵌套的函数调用

彼此必须相互合一或匹配的项允许包含子项, 这些子项实际上是那些必须在合一或匹配完成之前进行求值计算的表达式或函数调用。

这样的子项求值计算建立在需要的基础上。

在谓词调用中, 所有输入参数均在调用发生前进行计算, 所有输出参数均为变量, 不需要计算。

在匹配或合一能被确定之前, 子句头也可以包含必须计算的项。

(1) 所有不需要任何计算的匹配或合一在进行计算之前都被执行。

(2) 与输入参数相应的计算从左至右依次执行。在每次计算之后, 将其每个值与相应的输入进行比较。

(3) 计算子句体。

(4) 从左至右计算输出参数。

(5) 返回计算的值(如果谓词是一个函数)。

如果上述任意一步失败, 那么计算的其余部分都不能执行。

总的说来, 基本原则如下:

(1) 输入在另一匹配之后, 在子句体计算之前。

(2) 输出在子句体计算之后。

(3) 从左至右。

12.6.7 变量与常量

Prolog 中的变量是不可变的, 一旦被绑定了一个值就会保持该值, 除非利用回溯在恢

复先前程序状态的过程中重新释放这个变量。

一个变量可以在合一和匹配时被绑定，如果它已经被绑定，那么可以经过计算得出它所绑定的值。

变量名以大写字母或下划线开头，后面可以跟任意多个字母（大小写均可）、数字或下划线字符。例如，下面的变量名是合法的：

```
My_first_correct_variable_name
_
_Sales_10_11_86
```

下面两个变量名是非法的：

```
1stattempt
second_attempt
```

一个单独的下划线字符代表匿名变量。当一个变量的值没有作用时，可使用匿名变量。

如果一个变量在子句中只出现一次，并且变量名以下划线开始，那么 Visual Prolog 编译器就会把这个变量视为一个匿名变量。

Prolog 的变量是局部变量而不是全局变量。也就是说，如果两个子句分别都包含一个变量 X，那么它们是截然不同的两个变量。如果它们在合一中同时出现，就会互相绑定，但是通常不会互相影响。

当一个变量还没有与一个项关联起来，或一个项合一时未被绑定或初始化，就可以称之为自由变量。

Visual Prolog 编译器不区分名称中除首字母外字母的大小写。也就是说，如 SourceCode 和 SOURCECODE 是一样的。

一个常量（constants）表示它被定义的值。

12.6.8 算术表达式

算术表达式（arithmetic expressions）按算术运算符中所描述的运算符级别进行分析。对于二元表达式先计算左边的项，再计算右边的项。

（1）相等（=）

首先计算等式左边的项，然后计算右边的项，最后它们相互合一。合一常常可以简化为匹配。

（2）比较

首先计算左边的项，然后计算右边的项，最后比较结果。

注意：不等号（<或>）不是等号（=）的对偶运算。<比较两个值，而=是将两项合一（至少在大多数情况下）。

表达式 A=B 的对偶表达式为 not(A=B)。

12.6.9 事实断言与撤销

事实变量 (fact variables) 是可变变量，它们计算已经赋予它们的值。

一个新值可以通过赋值的方法赋给一个事实变量。

一个事实数据库包括若干完全用具体例证说明的谓词头，该谓词头与来自事实段定义的事实相对应。这些事实可以通过用事实名作为谓词名的谓词调用来访问。该谓词调用依次与每个事实相匹配；每当该谓词调用与事实相匹配时，后面就会跟一个下一事实可能的回溯点。当事实数据库中不再有事实时，该谓词调用失败。

新的事实可以用谓词 `assert/1`, `asserta/1` 和 `assertz/1` 进行断言。`assert/1` 与 `assertz/1` 一样，在事实列表的尾部断言一个新事实。`asserta/1` 在该列表的头部断言一个新事实。

已有的事实可用谓词 `retract/1` 和 `retractAll` 撤销。`retract/1` 撤销第 1 个在参数中与参数绑定变量匹配的事实，并留下一个回溯点以使更多的事实在回溯时可被撤销。

`RetractAll` 撤销所有与参数匹配的事实，无需任何绑定即可成立。

12.6.10 失败谓词和成功谓词

失败谓词 `fail/0` 和成功谓词 `succeed/0` 是两个内建的空变元谓词。`fail/0` 总是失败，`succeed/0` 总是成功，此外它们并无任何影响。

12.6.11 逻辑与

逻辑“与 (and)”，在程序的子句体中常用逗号表示。一个 `and` 表达式 “A, B” 按如下步骤进行处理。首先计算左子项 A。如果这一计算失败，整个 `and` 项失败。如果 A 成功则接着计算右子项。如果计算失败，则整个 `and` 项失败，否则 `and` 项成功。

因而，只有在第 1 子项 A 成功的情况下，第 2 子项 B 才会得到计算。

举例：

```
clauses
  ppp() :-
    qq(), rrr().
```

当 `ppp` 被调用时，它首先调用 `qqq`，如果 `qqq` 成功，则调用 `rrr`。如果 `rrr` 成功，则 `and` 项成功，随后即整个子句成功。

12.6.12 逻辑或

逻辑“或 (or)”，在程序的子句体中常用分号表示。一个 `or` 表达式 “A; B” 按如下步骤进行处理。首先创建一个指向第 2 项 B 的回溯点，然后计算第 1 项 A。如果第 1 项计算成功，则整个 `or` 项成功，留下一指向第 2 项 B 的回溯点。如果第 1 项的计算失败，则指

向第 2 项的回溯点被激活。

如果指向第 2 项的回溯点被激活（因第 1 项失败或在后面执行的某项操作中调用了回溯点），则第 2 项 B 被计算，若 B 成功，则整个 or 项成功。

因而一个 or 项可以在一个回溯点上获得成功，而第 2 子项 B 只在回溯过程中得到计算。

注意：现在 or 项仅能用在子句体的最外层。以后版本的 Visual Prolog 将改变这一点。

举例：

```
clauses
    ppp() :-
        qqg(V), write(V), fail.
    qqg(X) :-
        X = 3 ; X = 7.
```

当 ppp 被调用时，它首先调用 qqg。

qqg 先创建一个指向 X=7 的回溯点，然后计算 X=3。因此 X 将绑定为 3，而 qqg 将返回。因此 ppp 中的 V 绑定为 3，V 被打印输出为 3，然后出现 fail。fail 总是失败，因此控制权被设定在 qqg 的回溯点上。

因为创建了回溯点，回溯也使所有变量绑定失效。在这种情况下，意味着 ppp 中的 V 及 qqg 中的 X 都是非绑定变量。

然后，X=7 在 qqg 中得到计算，X 因而绑定到 7，控制回到 ppp，X 绑定到 7 并被打印输出。接着再次出现失败，这次 ppp 中没有更多的回溯点，所以 ppp 失败。

12.6.13 逻辑非

逻辑“非 (not)”，not/1 以一个项作为参数。对 not(A)的计算首先计算 A。如果 A 成功，则 not(A)失败，如果 A 失败，则 not(A)成功。

注意：not(A)不绑定任何变量，因为若 not(A)成功，则 A 失败，一个失败的变量不绑定任何量；另一方面，若 not(A)失败，它同样不能绑定任何变量，因为该项本身失败。

另外，not(A)也不能跟任何回溯点，因为若 not(A)成功，则 A 失败，而一个失败的项不能包含任何回溯点。这也意味着 A 中所有成功的可能性都没有了。

12.6.14 截断

截断 (cut) 在程序的子句体中常用感叹号 “!” 表示。截断会删除从当前谓词入口处开始创建的所有回溯点，这是指所有指向后来子句的回溯点，再加上截断之前的当前子句的谓词调用中安置的回溯点。

举例:

```
clauses
  ppp(X) :-
    X > 7,
    !,
    write("Greater than seven").
  ppp(_X) :-
    write("Not greater than seven").
```

当 ppp 被执行时，有一个首次创建的指向第 2 子句的回溯点，然后第 1 子句被执行。如果测试 “X>7” 成立，则截断 (“!”) 是可到达的。这一截断 “!” 将取消指向第 2 子句的回溯点。

举例:

```
clauses
  ppp() :-
    qqg(X),
    X > 7,
    !,
    write("Found one").
  ppp() :-
    write("Did not find one").
clauses
  qqg(3).
  qqg(12).
  qqg(13).
```

当 ppp 被执行时，它首先创建一个指向第 2 个 ppp 子句的回溯点，然后调用 qqg。qqg 将创建一个指向第 2 个 qqg 子句的回溯点并执行第 1 子句，从而返回值 3。在 ppp 中，变量 X 绑定为 3 这个值然后与 7 比较。测试失败，因而控制回溯至 qqg 的第 2 子句。

在执行第 2 子句前，指向 qqg 的第 3 子句的回溯点被创建，然后第 2 子句返回 12。

这次与 7 的比较测试成立，因而执行截断。这一截断将删除 qqg 中留下的回溯点和指向 ppp 中第 2 子句的回溯点。

12.6.15 谓词 finally/2

finally/2 是一个特殊的内部谓词，它以两个子项作为参数。无论第 1 项的计算结果如何，它确保对第 2 项的计算。如果满足以下条件，则必然计算第 2 项：

- 第 1 项成功
- 第 1 项失败

- 第 1 项意外终止

整个项表现如下：

- 如果第 1 项成功，那么整个项的表现就像用 `and` 连接了两项一样；
- 如果第 1 项失败，则整个项失败（在完成第 2 项的计算之后）；
- 如果第 1 项意外终止，则整个项也将意外终止（在完成第 2 项的计算之后）。

如果第 2 项意外终止，则整个项将意外终止。

第 1 个子项不能留下回溯点，也就是说，不可能回溯到 `finally/2` 谓词的第 1 部分。

举例：

```
clauses
    ppp() :-
        Resource = allocateExternalResource(),
        finally(
            use(Resource),
            deallocateExternalResource(Resource )).
```

这个例子是 `finally/2` 谓词的典型应用。不论资源使用的结果如何，外部分配的资源必须被释放。

12.7 程序段

程序段（program sections）用来声明和定义作用域内的实体。

```
section :
    constantsSection
    domainsSection
    predicatesSection
    constructorsSection
    factsSection
    clausesSection
    conditionalSection
```

不是所有的段都能在各种作用域中出现。更为详尽的内容参考接口、类声明和类实现。条件段在条件编译中介绍。

本章小结

本章介绍 Visual Prolog 的程序元素，内容包括项（terms）、常量、谓词、子句、事实、运算、程序段等。

习题 12

1. Visual Prolog 程序元素包含哪些内容？分析每项的含义。
2. 阅读下面的程序，找出程序中的错误。

PREDICATES

```
likes_shopping(symbol)
has_credit_card(symbol,symbol)
bottomed_out(symbol,symbol)
```

CLAUSES

```
likes_shopping(Who):-
    has_credit_card(Who,Card),
    not(bottomed_out(Who,Card)),
    write(Who," can shop with the ",Card, " credit card.\n").
has_credit_card(chris,visa).
has_credit_card(chris,diners).
has_credit_card(joe,shell).
has_credit_card(sam,mastercard).
has_credit_card(sam,citibank).
bottomed_out(chris,diners).
bottomed_out(sam,mastercard).
bottomed_out(chris,visa).
```

goal

```
likes_shopping(Who).
```

第 13 章 编译单元

本章介绍 Visual Prolog 编译单元的有关内容，包括接口、类声明、类实现、各种类型转换、条件编译、异常处理、预处理程序指令等。

一个程序由若干编译单元组成。编译器分别编译这些编译单元。编译的结果是一个个目标文件。这些目标文件（可能还有其他文件）连接在一起形成项目的目标文件。一个程序必须确实包含一个目标段，它是程序的入口点。

在一个单元中所有的引用名被声明或被定义的情况下，一个编译单元必须自包含。在几个编译单元中，可以包括接口定义和类声明（定义或声明必须在包含它们的所有单元内一致）。然而类实现（定义）只能在一个单独的单元中被定义。每个被声明的项也必须在项目中被定义，但是一些项可以在程序库中定义，就是说它们不需要文本定义。

一个编译单元（可能用 `#include` 指令构成）是编译数据项的序列。

```
compilationUnit :  
    compilationItem-list-opt
```

一个编译数据项是一个接口、类声明、类实现、目标段，或者是在条件编译中所说的一个有条件的编译数据项。

```
compilationItem :  
    conditionalItem  
    interfaceDefinition  
    classDeclaration  
    classImplementation  
    goalSection
```

13.1 接口

本节介绍 Visual Prolog 的接口的有关概念，内容包括接口的基本概念、接口与对象、开放限定（open qualification）、支持限定（support qualification）等。

13.1.1 接口的基本概念

一个接口定义了一个命名的对象类型。接口可以支持其他接口。详细内容参见支持限定。在接口中声明的所有谓词都是接口类型对象的对象成员。

接口也是一个全局作用域，在其中可以定义常量和论域。这样，在一个接口中被定义的常量和论域不是该接口指示的类型的一部分（或具有该类型的对象）。

这样的论域和常量可以通过限定接口名 `interface::constant` 或使用开放限定由其他作用域引用。

```
interfaceDeclaration :  
    interface interfaceName scopeQualifications sections end interface  
interfaceName-opt
```

```
interfaceName :  
lowerCaseIdentifier
```

在构造器尾部的接口名 (`interfaceName`) (如果存在) 必须与构造器开始的接口名相同。作用域限定 (`ScopeQualifications`) 必须是下面的类型:

- 支持限定 (`supportsQualification`)
- 开放限定 (`openQualification`)

段 `sections` 必须是下面的类型:

- 常量段 (`constantsSection`)
- 论域段 (`domainsSection`)
- 谓词段 (`predicatesSection`)
- 接口谓词段 (`predicatesFromInterface`)
- 条件段 (`conditionalSection`)

所有包含在条件段的部分也必须是这些类型。

13.1.2 接口与对象

如果一个接口没有明确地支持任何接口, 那么它就隐含地支持内部接口对象。

对象是一个空接口, 即它不包含谓词等内容。

对象的目的是作为所有对象的通用基本类型。

13.1.3 开放限定

开放限定 (`open qualification`) 可以更方便地引用类层次的实体。开放段把一个作用域名代入另一作用域, 以使这些名字可以在不受限制的情况下被引用。

开放对于对象成员的名字没有影响, 因为无论如何它们只能被一个对象访问。但是类成员名、论域、算符和常量可以不受限制地被访问。

当名字以这样的方式被带进一个作用域时, 可能会出现有些名字变得不明确。

开放段只会在它们所出现的作用域内产生影响。尤其是指在一个类声明中的开放段不会影响类实现。

```
openQualification :  
    open scopeName-comma-sep-list
```


13.1.4 支持限定

支持限定（`supports qualification`）只能在接口定义（`interfaceDefinition`）和类实现（`classImplementation`）中使用。

支持限定用于以下两种情况：

- 指定一个接口 A 扩展到另外一个接口 B，因此对象类型 A 是对象类型 B 的子类型。
- 声明一个特定类的对象“私下”具有比一个指定作为构造类型的类更多的对象类型。

支持存在一个传递关系：如果接口 A 支持接口 B，并且接口 B 支持接口 C，那么接口 A 也支持接口 C。

如果一个接口没有明确支持任何接口，那么就暗指它支持预定义的接口对象。

就功能而言，一个接口是否会支持一个特定的接口一次还是多次（直接或者间接）并没有差别，但是对于对象而言却有客观的差别。

当支持用于一个类的实现中时，结果是 `This` 不但可以与构造类型一起使用，而且还能与任何私有的所支持的对象类型一起使用。

```
supportsQualification :
    supports interfaceName-comma-sep-list
```

支持限定（`supportsQualification`）只能在接口定义（`interfaceDefinition`）和类实现（`classImplementation`）中使用。

如果接口有冲突的谓词，则它们就不能在一个支持限定内一起使用。

如果谓词具有相同的名字和变元数，具有不同的原始接口，那就是冲突的。

一个谓词的原始接口是该谓词文字上被声明的谓词的接口，同时它反对由支持限定间接声明的接口。

因此如果同一接口在支持链中出现两次或更多次，它也不会发生冲突。

考虑下面的定义和声明：

```
interface aaa
    predicates
        insert : (integer X) procedure (i)
end interface
interface bbb
    supports aaa
    predicates
        insert : (integer X, string Comment) procedure (i,i)
end interface
interface cc
    supports aaa
    predicates
        extract : () -> integer procedure ()
```

```

end interface
interface dd
  supports aaa, bbb, cc
  predicates
    extract : (string Comment) -> integer procedure (i)
end interface

```

这里是在 `dd` 中找到的所有谓词的列表（通过深度遍历查找）：

```

predicates
insert : (integer) procedure (i).           % dd -> aaa
insert : (integer) procedure (i).           % dd -> bbb -> aaa
insert : (integer, string) procedure (i,i). % dd -> bbb
insert : (integer) procedure (i).           % dd -> cc -> aaa
extract : () -> integer procedure ().        % dd -> cc
extract : (string) -> integer procedure (i). % dd

```

其中有些谓词是一样的，因此总的来说，`dd` 将包含下列成员：

```

predicates
insert:(integer) procedure (i).           %origin interface: aaa
insert:(integer, string) procedure (i,i). %origin interface: bbb
extract:() -> integer procedure ().        %origin interface: cc
extract:(string) -> integer procedure (i). %origin interface: dd

```

考虑下面的接口：

```

interface aaa
  predicates
    insert:(integer X) procedure (i).
end interface
interface bbb
  predicates
    insert:(integer X) procedure (i).
end interface
interface cc
  supports aaa, bbb % conflicting interfaces
end interface

```

接口 `cc` 是非法的，因为在 `aaa` 中所支持的 `insert/1` 以 `aaa` 为源，在 `bbb` 中所支持的 `insert/1` 则是以 `bbb` 为源的。

13.2 类声明

一个类声明（`class declarations`）定义针对环境的类的外部特征：环境完全可以看见和使用那些在类声明中提及的实体。类的声明指定了类的公共部分。

一个类声明可以包含常量和论域的定义，以及谓词的声明。

如果一个类声明了一个构造类型，那么它就构造这种类型的对象。构造类的对象至少有一个构造器（**constructor**），也可以定义得更多。不显式定义构造器的类会自动配置默认的构造器，即 **new/0**。

对象通过调用类的一个构造器来构造。

当初始化继承类时也用构造器。

在类声明中所提到的一切都属于该类，而不属于它所构造的对象。与该对象相关的一切，必须在该类所构造的对象的构造类型中被声明。

任何一个类声明必须伴有一个类实现。类实现提供了在类声明中所声明的谓词的定义或实现程序。同样，类实现还提供了由类构造的对象所支持的谓词的定义。子句可以实现两种类型的谓词，但是对象谓词也可以继承其他的类。

特别值得注意的是，一个类声明并不描述任何与代码继承有关的内容。代码继承是一个完全私有的事件，它只能在类实现中被声明（这不像其他的面向对象程序设计语言，它在实现中隐藏所有的细节）。

如果该类不声明一个构造类型，那么它就不能构造任何对象。因此它只能作为一个模块，而不能作为一个真正的类发挥作用。

```
classDeclaration:
    class className constructionType-opt scopeQualifications sections end class
    className-opt
```

```
constructionType :
    :interfaceName
```

```
className :
    lowerCaseIdentifier
```

在类声明尾部的类名（**className**）（如果存在的话）必须与其头部的类名一致。

注意，可以使用与指定为该类的构造类型的接口名构造类型（**constructionType**）相同的类名（**className**）。写作：

```
class interfaceAndClassName : interfaceAndClassName
```

注意：类和接口可以声明域和常量，并且由于它们在同名的空间内结束，所以一定不会发生冲突（因为它们只能以相同的接口名或类名限定）。

作用域限定（**scopeQualifications**）必须是开放限定（**openQualification**）类的。

段（**sections**）必须是下面的几种：

- 常量段（**constantsSection**）
- 论域段（**domainsSection**）
- 谓词段（**predicatesSection**）
- 构造段（**constructorsSection**）
- 条件段（**conditionalSection**）

构造段（`constructorsSections`）只在该类声明为构造类型（`constructionType`）时才是合法的。

所有包含在条件段中的部分必须也是上述种类。

13.3 类实现

本节介绍类实现（`class implementations`）的有关知识，包括类实现的基本概念、继承限定（`inherits qualification`）、归结限定（`resolve qualification`）、委托限定（`delegate qualification`）、`This` 修饰、构造（`construction`）、终结（`finalization`）等内容。

13.3.1 类实现的基本概念

类实现用于提供类声明中所声明的谓词的定义和构造器，以及它构造的对象所支持的任意谓词的定义。

类可以私有地（即在实现内部）声明和定义比类声明中提到的更多的实体。特别地，一个实现可以声明用于实现类和对象声明的事实数据库。

实现是一个混合作用域，在这个意义上来说，它包括了类的实现和类所产生的对象。类中的类部件在类的所有对象间共享，与对象部件相反，对象部件相对每个对象来说是单独的。类部件和对象部件都可以包含事实和谓词，而论域、算符和常量总是属于类部件，就是说它们不属于单个对象。

默认时，类实现中声明的所有谓词和事实对象都是对象成员。为了声明对象成员，段关键字（即 `predicates` 和 `facts`）前必须加上前缀 `class`。所有在这样的段中声明的成员都是类成员。

类成员可以引用类的类部件，但是不能引用对象部件。

另一方面，对象成员既可以访问类的类部件，又可以访问对象部件。

在实现的代码中，所有对象谓词都包含宿主对象。包含的宿主对象也可以直接通过特殊变量 `This` 来访问。

```
classImplementation :  
    implement className scopeQualifications sections end implement className-opt
```

在类实现尾部的类名（`className`）（如果存在的话）必须与其头部的类名一致。

作用域限定（`ScopeQualifications`）必须是下面的几种：

- 支持限定（`supportsQualification`）
- 开放限定（`openQualification`）
- 继承限定（`inheritQualification`）
- 归结限定（`resolveQualification`）
- 委托限定（`delegateQualification`）

支持限定描述接口列表，这些接口由类实现私有地给予支持。委托限定把接口谓词或

对象的功能委托给对象谓词，它们可以作为事实变量被存储起来。

段必须是以下几种：

- 常量段 (constantsSection)
- 论域段 (domainsSection)
- 谓词段 (predicatesSection)
- 构造器段 (constructorsSection)
- 事实段 (factsSection)
- 子句段 (clausesSection)
- 条件段 (conditionalSection)

只有类的类名 (className) 声明了一个构造类型 (constructionType) 时，构造器段 (constructorsSections) 才是合法的。声明了一个构造类型 (constructionType) 的那些类也是对象构造器，可以构造所声明的构造类型的对象。

下面这个例子说明了类事实怎样在类的对象中被共享，而对象事实不被共享。

考虑接口 aa 和类 aa_class:

```
interface aa
  predicates
    setClassFact : (integer Value) procedure (i) .
    getClassFact : () -> integer procedure () .
    setObjectFact : (integer Value) procedure (i) .
    getObjectFact : () -> integer procedure () .
end interface
class aa_class : aa
end class
```

谓词的指针是这样一种指针，它们可以从各个类和对象事实中存取其值：

```
implement aa_class
  class facts
    classFact : integer := 0.
  facts
    objectFact : integer := 0.
  clauses
    setClassFact(Value) :-
      classFact := Value.
    getClassFact() = classFact.
  clauses
    setObjectFact(Value) :-
      objectFact := Value.
    get ObjectFact() = objectFact.
end implement aa_class
```

假设这个类考虑目标 goal:

```
goal
  A1 = aa_class::new(),
  A2 = aa_class::new(),
  A1:setClassFact(1),
  A1:setObjectFact(2),
  ClassFact = A2:getClassFact(),
  ObjectFact = A2:getObjectFact().
```

类事实在所有的对象中被共享,所以通过 A1 设置的类事实也会影响通过 A2 获得的值。因此, ClassFact 的值只有一个,就是通过 A1 设置的值。

另一方面,对象事实属于每个对象。所以,在 A1 中设置的对象事实不会影响 A2 中存储的值。因此, ObjectFact 的值是零,它是 A2 中初始化的事实值。

13.3.2 继承限定

继承限定用于声明一个实现对一个或多个类的继承。继承只影响类的对象部件。

从其他类继承的目的是继承那些类的行为。

当类 cc 继承了类 aa,那么类 cc 的对象将包含类 aa 的一个嵌入对象。

对象谓词可以被继承:如果类不执行它的某个对象谓词,但是它所继承的其中一个类执行了这些谓词,那么这些谓词就将用于当前类。

从其他类继承的类对于继承类没有任何特权:它只能通过构造器类型接口访问所嵌入的对象。

继承必须是明确的。如果类定义了谓词本身,那么不会出现含糊的问题,因为这个谓词定义是被显式表达的。如果只有一个继承类支持谓词,那么类所提供的定义也是明确的。但是如果两个或两个以上的类支持该谓词,那么类所提供的定义就是含糊的。在这种情况下,必须通过一个归结限定,使这种含糊性得到解决(参见归结限定)。

源自继承类的对象谓词可以从当前类的对象谓词直接调用,这是因为内嵌的子对象(sub object)被隐含地用作谓词所有者。类限定可以用于解决来自继承类对象谓词的调用模糊性。

```
inheritsQualification :
  inherits className-comma-sep-list
```

13.3.3 归结限定

正如其他地方讲过的那样,凡是与谓词调用有关的含糊问题,都可以通过使用限定名来避免。但是当涉及继承问题时就不成立了。考查下面的例子:

```
interface aa
  predicates
  p : () procedure () .
  ...
end interface
```

```
class bb_class : aa
end class
```

```
class cc_class : aa
end class
```

```
class dd_class : aa
end class
```

```
implement dd_class inherits bb_class, cc_class
end implement
```

在本例中,类 `bb_class` 和类 `cc_class` 哪个向类 `dd_class` 提供 `aa` 的实现是不确定的(注意:当说到一个类实现一个接口时,就是指这个类向这个接口中声明的谓词提供定义)。

当然有可能向 `dd_class` 的实现添加子句,这将有效地解决这项工作。比如考虑下面的子句,它会从 `bb_class` 中输出谓词 `p`:

```
clauses
  p() :- bb_class::p().
```

但是,用这一代码并没有真正继承 `bb` 的行为,实际上是向该类的 `bb_class` 部件委托了这项工作。

因此要解决这种含糊性问题(并使用真正的继承而不是委托),使用一个归结(`resolve`)段。归结段包含若干个解:

```
resolveQualification :
  resolve resolution-comma-sep-list
```

```
resolution :
  interfaceResolution
  predicateFromClassResolution
  predicateRenameResolution
  predicateExternallyResolution
```

归结限定用于从指定的源程序中确定实现。

(1) 谓词归结

```
predicateFromClassResolution :
  predicateNameWithArity from className
```

一个来自类归结(`class resolution`)的谓词,声明该谓词由指定的类实现。

为了归结一个类的谓词:

- 该类必须实现待归结的谓词,意指该谓词必须源于本应继承的同一谓词。
- 该类必须在继承段中提到。

(2) 谓词重命名归结

谓词重命名归结(`predicate rename resolution`)声明一个谓词以另外一个名字被实现。

该谓词必须来自一个继承类，并且它的类型、模式和流必须确实匹配。

```
predicateRenameResolution :
    predicateNameWithArity from className :: predicateName
```

(3) 接口归结

接口归结 (**interface resolution**) 用于归结来自继承类的完整接口。这样，一个接口归结就成为声明接口中所有的谓词应当由同一类归结的一个捷径。

```
interfaceResolution :
    interface interfaceName from className
```

该类必须公有地支持被归结的接口。

如果谓词归结和接口归结都包含某一谓词名，那么就用谓词归结。也就是说，特定的归结会覆盖不太具体的归结。

一个谓词被几个接口归结覆盖是合法的，只要那些接口归结都能将谓词归结到同一类中。另一方面，如果一个谓词被接口归结到不同的类，那么结果的含糊性就必须用谓词归结来解决。

注意：归结语法不能将一个谓词的不同重载放到不同的类中。

可以通过提供一个接口归结方法，解决上述例子中的含糊性。

在这种情况下，选择从 `cc_class` 继承 `aa_class` 的实现，此外还从 `bb_class` 继承 `p`。

```
implement dd_class
    inherits bb_class, cc_class
    resolve
        interface aa from cc_class
        p from bb_class
end implement
```

(4) 外部归结

一个谓词的外部归结 (**external resolution**) 声明该谓词并没有在类本身中被实现，而是在外部程序库中被实现的。外部归结只能用于类谓词，即对象谓词不能被外部归结。

在程序库中，调用约定、连接名和参数类型要与实现相符是非常重要的。

私有的和公有的谓词都能被外部归结。

```
predicateExternallyResolution : predicateNameWithArity externally
```

(5) 动态外部归结

一个谓词的外部归结也向来自 DLL 的私有和公共类谓词的动态载入提供语法。该语法如下：

```
predicateExternallyResolutionFromDLL :
    predicateNameWithArity externally from DllNameWithPath
```



```
DllNameWithPath :
    stringLiteral
```

如果谓词 `predicateNameWithArity` 在 DLL `DllNameWithPath` 中被丢失，那么动态载入就给运行程序提供了可能性，直到实际调用这个被丢失的谓词为止。在这样的调用中将出现运行时错误。

13.3.4 委托限定

一个委托段包括许多的委托：

```
delegateQualification :
    delegate delegation-comma-sep-list
```

```
delegation :
    predicateDelegation
    interfaceDelegation
```

委托限定用于将对象谓词的执行委托给指定的源。

有两种委托限定，即谓词委托和接口委托。接口委托，用于将一个接口声明的一整套对象谓词的实现委托给另一个作为事实变量存储的对象的实现。这样，一个接口委托就成为声明以下内容的一个捷径，即接口中所有谓词的实现应当委托给以事实变量存储的对象的实现。

委托段除了它是委托给那些保持类的构造对象的事实变量而不是委托给继承类之外，它看起来像是归结段的对应物（谓词或接口）。

(1) 谓词委托

对象谓词委托（`predicate delegation`）声明该谓词的功能被委托给以事实变量 `factVariable_of_InterfaceType` 所指定的对象中的谓词。

```
predicateDelegation :
    predicateNameWithArity to factVariable_of_InterfaceType
```

将一个谓词委托到一个以事实变量传递的对象：

- 事实变量 `factVariable_of_InterfaceType` 必须有一个接口的类型(或其子类的类型)，它声明谓词 `predicateNameWithArity`。
- 支持接口的对象必须被构造出来并被赋值给事实变量 `factVariable_of_InterfaceType`。

考查下面的例子：

```
interface a
    predicates
        p1 : () procedure () .
        p2 : () procedure () .
end interface
```

```

interface aa
    supports a
end interface

```

```

class bb_class : a
end class

```

```

class cc_class : a
end class

```

```

class dd_class : aa
    constructors
        new : (a First, a Second).
end class

```

```

implement dd_class
    delegate p1/0 to fv1, p2/0 to fv2
    facts
        fv1 : a .
        fv2 : a .
    clauses
        new(I,J):-
            fv1 := I,
            fv2 := J.
end implement

```

随后可以构造类型 `a` 的对象，并把它们赋给事实变量 `fv1` 和 `fv2` 来定义对象，从而定义真正委托 `p1` 和 `p2` 功能性定义的类的对象。考虑下列例子：

```

goal
    O_bb = bb_class::new(),
    O_cc = cc_class::new(),
    O_dd = dd_class::new(O_bb, O_cc),
    O_dd : p1(),          % This p1 from O_bb object
    O_dd : p2().          % This p2 from O_cc object

```

实际上，Visual Prolog 的委托具有与向 `dd_class` 的实现添加子句一样的效果，`dd_class` 由谓词的功能是“输出”的类的对象明确指定。例如，好像在 `dd_class` 的实现中确定了下面的子句：

```

clauses
    p1() :- fv1 : p1().

```

(2) 接口委托

当需要指定在接口 `interfaceName` 中声明的所有谓词的功能被委托到来自同一被继承类的对象的谓词时，可以使用接口委托说明：

```
interfaceDelegation :
    interface interfaceName to factVariable_of_InterfaceType
```

这样接口委托就是一个捷径，声明了接口 `interfaceName` 中声明的所有谓词的功能应当被委托给以事实变量 `factVariable_of_InterfaceType` 存储的对象。对象被赋以事实变量 `factVariable_of_InterfaceType`，该变量是 `interfaceName` 类型（或其子类型）的。

委托一个接口给一个通过事实变量传递的对象：

- 事实变量 `factVariable_of_InterfaceType` 必须具有一个接口 `interfaceName` 的类型或其子类型。
- 对象所支持的接口必须被构造出来并被赋值给事实变量 `factVariable_of_InterfaceType`。

谓词委托比接口委托优先级更高。如果一个谓词的两种委托都被指定，即谓词委托被指定到谓词，并且在具有接口委托的接口中被声明，那么具有较高优先权的谓词委托将被执行。

13.3.5 This 修饰

对象谓词总是在一个对象中被调用。这个对象携带该对象的事实，并且包含在对象谓词的实现中。对象谓词有权使用隐含的对象。这样的对象被称为 **This**。有两种 **This** 的访问，即隐式访问和显式访问。

1. 显式 This

在每个对象谓词的每个子句中，变量 **This** 被隐含地定义并绑定到 **This**，也就是其成员谓词正在执行的对象。

2. 隐式 This

在一个对象成员谓词子句中，可以直接调用另一对象成员谓词，因为 **This** 是被隐含地包含的运算。只要调用的方法含义明确，超类的成员也可以被直接调用（参见作用域和可视性）。同样地，对象事实（存储在 **This** 中）也可以被访问。

3. This 和继承

注意，这个具体部分将被改变，因为已经决定要改变该问题的语言语义。

This 总是引用一个属于用 **This** 的类的对象，如果一个类被另一个类继承也成立。

假定接口 `aa` 被声明如下：

```
interface aa
    predicates
        action : () procedure ().
        doAction : () procedure ().
end interface
```

再假定类 `aa_class` 如下：

```
class aa_class : aa
end class
```

然后执行:

```
implement aa_class
  clauses
    action() :-
      doAction(),
      This:doAction().
    doAction() :-
      write("aa_class::doAction"), nl.
end implement
```

下面的目标:

```
goal
  A = aa_class::new(),
  A:action().
```

将输出:

```
aa_class::doAction
aa_class::doAction
```

现在再考虑类 `bb_class`, 其声明如下:

```
class bb_class : aa
end class
```

其实现如下:

```
implement bb_class inherits aa_class
  clauses
    doAction() :-
      write("bb_class::doAction"), nl.
end implement
```

下列目标:

```
goal
  B = bb_class::new(),
  B:action().
```

也将输出:

```
aa_class::doAction
aa_class::doAction
```

这是因为在隐式和显式情况下，`aa_class` 中的 `This` 都引用了类 `aa_class` 的对象。

13.3.6 构造器

本节介绍对象的构造，同样它只涉及产生对象的类。通过调用构造器构造对象。构造器在类声明和类实现的构造器段中明确地声明（参见缺省构造器）。

一个构造器实际上有两个相关的谓词：

- 一个类函数，返回新构造好的对象。
- 一个对象谓词，当初始化继承对象时使用。

这个相关的对象谓词用于执行该对象的初始化。这个谓词只能从类自身的构造器中，以及从该类的继承类的构造器中被调用（即基本类初始化）。

这个相关的类函数是隐含定义的，即任何地方都不存在它的子句。

类函数分配内存来保留该对象，并执行该对象的内部初始化；然后它再调用被创建的对象的对象构造器；最后，被创建的对象作为结果返回。

因此在构造器的子句被调用之前：

- 所有具有初始化表达式的对象事实变量都被初始化；
- 所有具有子句的对象事实从这些子句中被初始化。

在该构造器的子句被调用之前，这一初始化也在所有继承子对象中进行。

构造器的子句必须：

- 对所有单个事实变量和进入前未经初始化的对象事实变量进行初始化。
- 初始化所有的子对象。

构造器子句也可以不做其他的事情，但它必须完成这里所说的初始化任务以确保对象在构造之后是有效的。

注意：在构造对象的过程中有可能出错，要特别注意不要访问未经初始化的对象部分（参见构造对象的规则）。

1. 默认构造器

默认构造器是一个名为 `new` 的空变元构造器。如果一个类根本就没有声明任何构造器，那么它就隐含地声明了默认构造器。也就是说所有子句至少具有一个构造器。显式地重新定义默认构造器也是合法的。

不需要定义（即实现）默认构造器；如果它没有定义，则一个无效的定义隐含地被假定。假定一个接口 `aa`，考查下面代码：

```
class aa_class : aa
end class
```

类 `aa_class` 没有声明构造器；因此，它隐含地声明了默认构造器。这样可以创建一个如下的类 `aa_class` 对象：

```
goal
_A = aa_class::new(). % implicitly declared default constructor
```

实现类 `aa_class` 隐含声明的默认构造器是合法的：

```
implement aa_class
  clauses
    new() :-
      ...
  end implement
```

这个类声明了一个非默认的构造器，那么这个类就不会有默认的构造器。

```
class bb_class : aa
  constructors
    newFromFile : (file File).
  end class
```

这个类声明了一个构造器并且它也声明了默认构造器；因此很明显，它有一个默认构造器。

```
class cc_class : aa
  constructors
    new : ().                % default constructor
    newFromFile : (file File).
  end class
```

2. 子对象构造

所有的构造器都负责将被构造的对象初始化为有效状态。为了获得这样的有效状态，所有的子对象（即继承类）也必须被初始化。

子对象可以通过两种方式中的任意一种来初始化。这两种方式或者是程序调用一个继承类的构造器，或者是自动调用默认的构造器。后者实际上要求继承类具有默认的构造器。

如果继承类不具有默认构造器，那么必须显式地调用其他的构造器。在具有子句的事实和事实变量的初始化之后，在进入构造器的子句之前，必须立即执行继承类的构造器的默认调用。

通过没有返回值的形式调用继承类的构造器。如果调用是有返回值的形式，实际上是创建了一个新的对象，而不是调用 `This` 中的构造器（参见下面的例子）。

类 `bb_class` 的实现继承了类 `aa_class`，并且 `bb_class` 的默认构造器用一个最新创建的 `cc_class` 对象来调用 `aa_class` 的构造器。

```
implement bb_class inherits aa_class
  clauses
    new() :-
      C = cc_class::new(),      % create a cc_class object
      aa_class::newC(C).        % invoke constructor on inherited sub-object
    ...
  end implement
```

3. 单个（对象）事实初始化

正如所有的构造器必须初始化或构造子对象一样，它们也必须在首次引用对象的单个事实之前，对所有这些事实进行初始化。

注意：单个类事实只能用于子句初始化，因为它们与对象无关。一个类事实可以在创建第 1 个对象之前被访问。

下面的例子表明：

- (1) 怎样通过一个表达式初始化一个事实变量；
- (2) 怎样通过一个子句初始化一个单个事实（点）；
- (3) 怎样在构造器中初始化一个单个事实；
- (4) 在哪里调用一个继承类的默认构造器。

```
implement bb_class inherits aa_class
  facts
    counter : integer := 0.
    point : (integer X, integer Y) single.
    c : (cc C) single.
  clauses
    point(0, 1).
    % The object is created and counter and point are initialized before entrance,
    % the default constructor aa_class::new/0 is also invoked before entrance
    new() :-
      C = cc_class::new(),
      assert(c(C)).
    ...
end implement
```

4. 委托构造

作为用构造器直接构造对象的选择方案，可以将这一任务委托给同一个类的另一构造器。这一点仅仅通过调用另外一个构造器就可以实现（即不返回值的形式）。委托构造的时候必须确保已经实际构造了对象，并且不是“重复构造（over-constructed）”。单个事实可以被赋值任意多次，因此它不能“重复构造”。另一方面，继承类在对象构造期间只能被初始化一次。

下面的例子说明了一个以委托方式构造的典型应用。一个构造器（new/0）用默认值调用另外一个构造器（newFromC/1）。

```
implement aa_class
  facts
    c : (cc C) single.
  clauses
    new() :-
```

```

    C = cc_class::new(),
    newFromC(C).
newFromC(C) :-
    assert(c(C)).
...
end implement

```

5. 构造对象的规则

程序员必须确保：

- 所有子对象都只能被初始化或构造一次。
- 所有单个事实都应被初始化至少一次。
- 在子对象被初始化或构造之前，没有子对象被引用。
- 在单个事实被初始化之前没有被使用。

在编译时间内编译器能否检测出上述问题并不能得到保证。

编译器可以提示产生一个运行时间确认，它也可以提示非安全地省略这些运行时间确认。

13.3.7 终结

一旦程序不能到达一个对象，这个对象就被终结。语义上并不能确切地指出这个对象何时会终结。惟一能够确定的是，只要程序还能到达，对象就不会终结。习惯上，当对象被垃圾回收程序废弃的时候，它就终结了。终结是构造的对立面，它将对象从内存中删除。

类也可以实现一个终结器，它是一个对象终结时（从内存被删除之前）调用的谓词。

一个终结器是一个名为 **finalize**，没有参数也没有返回值的过程。这个谓词隐含地被声明，并且不能由程序直接调用。

终结器的主要用途是释放外部资源，但是对于它所能做的事情并没有限制。使用终结器时应当小心，它们的调用时间不能完全知晓，因此，在终结器被调用的时候很难预测出整个程序的状态。

注意：在终结器中撤销一个对象的对象事实是不容置疑的，因为这是由终结进程自动完成的。

在程序终止前，所有的对象都被终结（除非像电源故障这样的异常情况出现）。

这个例子用一个终结器确保一个数据库连接得到正确关闭：

```

implement aa_class
  facts
    connection : databaseConnection.
  clauses
    finalize() :-
      connection:close().
    ...
end implement aa_class

```


13.4 类型转换

本节介绍 Visual Prolog 的各种数据类型的转换，包括隐式转换、显式转换等内容。

13.4.1 隐式转换

隐式转换 (implicit conversion) 又称为自动转换，它涉及可视类型和构造类型等。下面将对此举例予以说明。

1. 可视类型和构造类型

一个接口定义了一个对象类型，所有支持该接口的对象都有这个类型。以后用对象类型作为术语表示由接口定义的类型同义词。

由于接口定义了类型，这些类型可以用作谓词和事实声明中的，以及论域定义中的形式类型说明符。

由于一个对象含有它所支持的任意接口的对象类型，所以它可以用作这些类型中任意一个类型的对象。也就是说，对象的类型可以转换为任意的所支持对象类型。正如下面所述，这种转换在大多数情况是自动执行的。

因此，同一对象在不同的上下文中可以视为具有不同的类型。一个对象可见的类型被称为可视类型 (view type)，而构造该对象的类的类型被称为构造类型 (construction type) 或定义类型 (definition type)。构造类型也是一个可视类型。

2. 类型转换

如上所述，对象可以用作具有任意接口支持的类型。这部分介绍如何处理各种类型之间支持的转换。

如果某一术语静态地代表某一类型 T1，并且 T1 被声明支持 T2，那么显然变量引用的任何对象都将真正支持 T2。因此，向上的支持信息被静态地告知。随后自动执行支持层次的所有向上转换。

假定支持接口 aa 的接口 bb 存在，并且类 bb_class 具有构造类型 bb。考查下面的代码：

```
implement ...
predicates
  ppp : (aa AA) procedure (i) .
clauses
  ... :-
    BB = bb_class::new(), % (1)
    ppp(BB).               % (2)
clauses
```

```

ppp(AA) :-                                % (3)
...
    BB = convert(bb,AA), % Conversion possible since definition type of
AA is bb
...

```

在标记为 (1) 的行中，创建了一个 `bb_class` 对象：该对象具有构造类型 `bb`。变量 `BB` 是这个新对象的引用。`BB` 提供了项目中这个对象的可视类型 `bb`。

在标记为 (2) 的行中，该对象作为一个 `aa` 对象传递给 `ppp`。隐含地执行从可视类型 `bb` 到可视类型 `aa` 的转换。当对象到达标记 (3) 的行时，虽然构造类型还是 `bb`，但它却具有可视类型 `aa`。

13.4.2 显式转换

显式转换 (explicit conversion) 通过调用一个转换谓词来执行。有几个转换谓词是可以利用的。

1. 受检查的转换

谓词 `convert//2` 和 `tryConvert//2` 用于实现一个类型到另一类型的安全转换。

任一个谓词都不能赋予真实的声明，但是这里有它们的伪声明：

```

predicates
convert : ("type" Type, _ Value) -> Type ConvertedValue.
tryConvert : ("type" Type, _ Value) -> Type ConvertedValue determ.

```

`convert//2` 和 `tryConvert//2` 都采用 `type` 作为第 1 个参数，一个任意类型的值作为第 2 个参数，并且随后返回已转换的值作为结果。

如果不可能转换的话，`convert//2` 将产生一个异常，而 `tryConvert//2` 只会失败。

注意：如果源类型是这个目标类型的子类型，那么使用 `convert//2` 和 `tryConvert//2` 就是多余的，因为转换将被隐含地完成。

`convert//2` 和 `tryConvert//2` 可以用于下面的情形：

- 从一个数值论域 (number domain) 转换到另一个数值论域；
- 从对象转换到另一类型。

如果可以确定转换永远也不会成功，那么编译器可以提出抗议（但不是必须的）。比如，试图在没有重叠部分的两个数值论域之间进行转换。

2. 向下转换

当一个对象被转换成一个超类型（即一个被支持的接口）时，关于对象的信息将被“遗忘”。注意，它的性能并不是真的被丢失，它们只是在具有较小能力接口的对象上下文中不可见。在多数情况下，需要恢复该对象的实际能力。因此，要既能够向上转换也能够向下转换。

向下转换通常不能静态地生效。因此，当恢复“丢失的”接口时，有必要用显式转换。

当在支持层内向上的类型转换的说明非常简单时，要求有一个“真实”的例子来说明向下转换的切合实际的应用。因此，这里引用一个更为“真实”的例子。

假定要执行“一些同种的对象”。也就是说，一组对象都支持同一个特定的可视类型。需要几组这样的对象类型，因此要以这样的方式来安排实现，即这些程序可以容易地被许多不同类型的对象所采用，但是仍然保留所包含对象的同种性。

这里的方法相当标准：使这些“组”的实际实现基于任意对象支持的对象类型。然后用细小的层构造更多具体的“组”，这些细小层可以在实际的类型和对象之间转换。这里不给出对象组的实际实现，仅仅假定它存在于下面的类和接口中：

```
interface objectSet
  predicates
    insert : (object Elem) procedure (i)
    getSomeElem : () -> object determ (i)
    ...
end interface
class objectSet_class : objectSet
end class
```

现在假定有某种对象类型 myObject，并且要建立相应的“组”类 myObjectSet_class。声明 myObjectSet_class 如下：

```
interface myObjectSet
  predicates
    insert : (myObject Elem) procedure (i).
    getSomeElem : () -> myObject determ ().
    ...
end interface
class myObjectSet_class : myObjectSet
end class
```

也就是说，myObjectSet 具有 objectSet 的所有谓词，但是每个出现的对象都被 myObject 替代。myObjectSet_class 的实现从 objectSet_class 继承，这个嵌入或继承的 objectSet 将携带这个组的成员。该实现将实现下列不变量（invariant）：嵌入的 objectSet 只能包含类型为 myObject 的对象（即使它们在技术上有类型对象）。

实现如下：

```
implement myObjectSet_class
  inherit objectSet_class
  clauses
    insert(Elem) :-
      objectSet_class::insert(Elem). % (1)
    getSomeElem() = Some :-
      SomeObject = objectSet_class::getSomeElem(), % (2)
```

```

        Some = convert(myObject, SomeObject).           % (3)
    ...
end implement

```

在标记为（1）的行，Elem 被自动地从类型 myObject 转换为对象。在标记为（2）的行，从嵌入的对象 object 组收回一个对象。这个对象在技术上有类型对象。但是由不变量知这个对象也支持 myObject。随后，便可以安全地恢复 myObject 接口。这一步在标记为（3）的行显式地完成。

3. 私有类型和公有类型

当一个对象用构造器建立时，它返回时就带有一个构造器类型。这样的对象可以自动地被转换为任意支持的接口并且再显式地转换回来。

即使执行对象的类已经声明了更多的私有支持接口，它也不可能将“公有”对象转换为任何私有对象。

但是在实现中，该对象可以被任意私自支持的类访问。此外，用任意这些私有支持的类可以在实现外处理 This。

这样的“私有”对象也可以在它的层次中被隐式地向上转换，然后显式地向下转换。实际上，这样的“私有”对象可以被显式地转换到任意公有的或私有的支持接口。

因此一个对象有两个视图（views）：公有视图和私有视图。私有视图包括公有类型。对象不能从一个视图转换到另一个视图，但是既然私有视图包括公有视图，那么私有视图就可以转换成任意支持的类型。

4. 无检查的转换

谓词 uncheckedConvert/2 用于执行基于存储器表示（memory representation）的非安全转换。该谓词不能以任何方式修改内存，它只是强制编译器用另一类型来解释存储段。

但该谓词非常不安全，使用时要特别警惕。

当与其他语言接口时，为了解释这些语言所用的存储映像，才使用该谓词。

uncheckedConvert/2 只能用于位长度完全相同的内存段。但是，一个指针代表了多种数据，并且这些数据位具有相同长度。

```

predicates
    uncheckedConvert : ("type" Type, _ Value) -> Type ConvertedValue.

```

举例：

```

predicates
    interpretBufferAsString : (pointer BufferPointer) -> string Value.
clauses
    interpretBufferAsString(BufferPointer) = uncheckedConvert(string, BufferPointer).

```

这个谓词把一个指针所代表的缓冲区解释为（转换为）一个字符串。当内存块有作为字符串的正确表示时，它才是合理的。

13.5 条件编译

条件程序设计结构是 Visual Prolog 语言的一部分（与预处理相对）。只有编译项和程序段可以是有条件的。

```
conditionalItem :  
    #if condition #then compilationItem-list-opt elseifItem-list-opt elseItem-opt  
    #endif
```

```
elseifItem :  
    #elseif condition #then compilationItem
```

```
elseItem :  
    #else compilationItem
```

```
conditionalSection :  
    #if condition #then section-list-opt elseifSection-list-opt elseSection-opt  
    #endif
```

```
elseifSection :  
    #elseif condition #then section-list-opt
```

```
elseSection :  
    #else section-list-opt
```

这里条件（condition）可以是任何表达式，这些表达式可以计算失败或者成功。

为了确定哪部分包含在最终的程序中，编译器在编译过程中计算条件。最终程序不包含的部分被称为死支。

条件编译项的所有分支都要经过语法检查，并且必须在语法上是正确的。也就是说，死支也应当在语法上是正确的。

编译器只在需要时计算条件，即它不计算死支上的条件。

条件在编译时间被计算，因此必须具有这种计算的可能性。

一个条件可以不依赖于任何位于条件语句内的文本代码。

13.6 异常处理

异常处理（exception handling）系统的基本部分基于两个内部谓词：errorExit/1 和 trap/3。

- errorExit 产生一个异常；
- trap 为一个特定计算设置一个异常处理器。

当 errorExit 被调用，当前活动的异常处理器就被调用。这个异常处理器在它原始的上下文中被执行，即在它被设置的上下文中，而不是在产生异常的上下文中被执行。

errorExit 所调用的参数被传送到异常处理器。这个参数必须提供所需的该异常的描述。

与附加的运行时间例程（additional runtime routines）一起有可能在该系统的顶端建立起高水平的异常处理机制。

但是，关于运行时间访问例程不在本节的讨论范围之内。同样，运行时间系统怎样处理发生在其内部的异常也不在本书讨论范围之内。

`trap` 的第 1 个参数是与新的异常处理器一起执行的条件。第 2 个参数必须是变量。如果在异常处理器活动的时候它被调用，那么这个变量就被绑定到 `errorExit` 被调用的值上。第 3 个变量是异常处理器，如果在异常处理器活动的时候 `errorExit` 被调用，则它将被调用。

这个异常处理器可以访问在第 2 个参数中声明的变量，从而检查所产生的异常。

举例：

```
clauses
  p(X) :-
    trap( dangerous(X), Exception, handleDangerous(Exception) ).
```

如果在执行 `dangerous` 时产生异常，那么 `Exception` 将被绑定到异常值，并且控制将转移到 `trap` 的第 3 个参数。在这种情况下，`Exception` 被传递给 `handleDangerous`。

13.7 预处理程序指令

预处理程序不是 Visual Prolog 语言的一部分。

每个编译指令以 `#` 字符开始。支持下面的编译指令。

- (1) 条件编译指令：`#if`, `#then`, `#else`, `#elseif`, `#endif`。
- (2) 源文件包含：`#include`。
- (3) 编译时间信息：`#message`, `#error`, `#requires`, `#orrequires`。

13.7.1 条件编译指令

条件编译编译器指令可以用于程序编译项和程序段（在条件编译一节中描述了它的语法），因此对于其他预处理程序指令也一样：

```
conditional_PP_Section :
  #if condition #then PP_Directive-list-opt elseif PP_Section-list-opt
  else PP_Section-opt #endif
```

```
elseif PP_Section :
  #elseif condition #then PP_Directive-list-opt
```

```
else PP_Section :
  #else PP_Directive-list-opt
```

```
PP_Directive :
  #include string_literal
```

```
#message string_literal
#requires string_literal
#orrequires string_literal
#error string_literal
```

条件（condition）必须是布尔表达式，在编译中可以求值计算。

每个条件编译语句 `conditional_PP_Section` 必须在同一文件中，也就是说，编译指令 `#if`，`#then`，`#elseif` 和 `#else` (如果存在的话)，以及同一嵌套层的 `#endif` 都必须在同一文件内。

13.7.2 源文件包含

`#include` 指令用于在编译中将另一文件的内容包含到程序的源代码中。它的语法如下：

```
pp_dir_include :
    #include string_literal
```

`string_literal` 应该指定一个存在的文件名。这个文件名可以包含一个路径名，但是必须记住用于给出子目录的反斜杠字符是一个换码符。基于此，当在直接写入源文本的路径中使用反斜杠时，必须使用两个反斜杠字符。

```
#include "pfc\exception\exception.ph"
        % Includes pfc\exception\exception.ph file
```

或者在文件名前加一个 `@` 作前缀：

```
#include @"pfc\vpi\vpimessage\vpimessage.ph"
        % Includes pfc\vpi\vpimessage\vpimessage.ph
```

这个指令使用“仅包含第 1 个出现的文件”的语义。也就是说，如果一个编译单元中对于同一文件，有几个包含指令，那么它就只包含一次，即第 1 个出现的指令。

每个被包含的文件必须包含一个或几个完整的作用域；一个被包含的文件不能包含不完整的作用域。也就是说，它应当包含一个或几个已完成的接口声明、类声明和类实现，或者几个预处理程序指令。

编译器以下列方式查找指定的包含文件：

- (1) 如果文件名包含一个绝对路径，那么这个文件就应当被包含；
- (2) 否则，编译器在当前目录中搜索指定的包含文件；

(3) 否则，编译器在由 `/Include` 命令行选项定义的路径中搜索指定的包含文件。因为这些路径在该选项中被指定，所以得到处理。在 `VDE` 中，可以在 `Project Settings` 对话框的 `Directories` 标签的 `Include Directories` 中设置这些路径。

13.7.3 编译时间信息

指令 `#message`，`#requires`，`#orrequires` 和 `#error` 用于在编译程序模块时发布用户定义的消息到一个列表文件，并中断编译。

这些指令既可以用于作用域（接口声明、类声明或类实现）的外部，也可以用于作用域的内部，但应在段的外部。其语法如下：

```
pp_dir_message :
    #message string_literal
```

```
pp_dir_error :
    #error string_literal
```

```
pp_dir_requires :
    #requires string_literal pp_dir_orrequires-list-opt
```

```
pp_dir_orrequires :
    #orrequires string_literal
```

当编译器遇到这些指令中的任何一个指令的时候，它将产生相应的警告消息，并将该指令文本放入一个列表文件。

一个列表文件名可以用下列编译指令来指定：

```
/ListingFile:"FileName"
```

注意：在冒号“:”（在/ListingFile 之后）与"FileName"之间没有空格。

默认时，编译器不为#message，#requires 和#orrequires 指令产生情报消息。可以通过下面指定的编译器选项，打开这些情报信息的产生功能。

```
/listing:message
/listing:requires
/listing:ALL
```

在这种情况下，当编译器遇到像这样的指令：

```
#message "Some message"
```

它就把下面的文本放入列表文件：

```
C:\Tests\test\test.pro(14,10) : information c062: #message "Some message"
```

指令#requires（#orrequires）向列表文件发布任意的用户自定义的有关所需要的源（对象）文件的消息。#orrequires 指令不能单独使用，#requires 指令应该在它的紧前面使用（只用空格或注释分隔）。

#error 指令总是使编译终止，并向列表文件发布如下用户定义的错误消息：

```
C:\Tests\test\test.pro(14,10) : error c080: #error "Compilation is interrupted"
```

可以分析这些消息并接受所要求的动作。比如，VDE 分析由指令#requires 和#orrequires 显示的消息，并自动向被编译的项目添加所有需要的 PFC 包和标准外部程序库（参见处理项目模块的相关主题）。

#requires 和 #orrequires 指令的例子如下：


```
#requires @"Common\Sources\CommonTypes.pack"
#orrequires @"Common\Lib\CommonTypes.lib"
#orrequires @"Common\Obj\Foreign.obj"
#if someClass::debugLevel > 0 #then
    #requires @"Sources\Debug\Tools.pack"
    #orrequires @"Lib\Debug\Tools.lib"
#else
    #requires @"Sources\Release\Tools.pack"
    #orrequires @"Lib\Release\Tools.lib"
#endif
#orrequires "SomeLibrary.lib"
#requires "SomePackage.pack"
#if someClass::debugLevel > 0 #then
    #orrequires @"Debug\SomePackage.lib"
#else
    #orrequires @"Release\SomePackage.lib"
#endif
```

#message 指令的例子:

```
#message "Some text"
#if someClass::someConstant > 0 #then
    #message "someClass::someConstant > 0"
#else
    #message "someClass::someConstant <= 0"
#endif
class someClass
    #if ::compiler_version > 600 #then
        #message "New compiler"
        someConstant = 1.
    #else
        #message "Old compiler"
        someConstant = 0.
    #endif
end class someClass
```

#error 指令的例子:

```
#if someClass::debugLevel > 0 #then
    #error "Debug version is not yet implemented"
#endif
```

本章小结

本章介绍 Visual Prolog 编译单元的有关内容, 包括接口、类声明、类实现、各种类型转换、条件编译、异常处理、预处理程序指令等。

一个程序由若干编译单元组成。编译器分别编译这些编译单元。编译的结果是多个目标文件。这些目标文件（可能还有其他文件）连接在一起形成项目的目标文件。一个程序必须确实包含一个目标段，它是程序的入口点。

习题 13

1. 理解接口的基本概念。为什么要引入接口声明？
2. 分析下列程序段中的非法接口。

```
interface aaa
  predicates
    insert : (integer X) procedure (i).
end interface
interface bbb
  predicates
    insert : (integer X) procedure (i).
end interface
interface cc
  supports aaa, bbb % conflicting interfaces
end interface
```

3. 作用域限定中的支持限定、开放限定、继承限定、归结限定和委托限定分别是什么含义？
4. Visual Prolog 中的 This 修饰与其他高级程序设计语言中的 This 有什么区别吗？

第 14 章 内部论域、谓词和常量

本章包括所有内部常量、论域和谓词的声明和描述。

14.1 概述

Visual Prolog 包含嵌入式隐藏类，它提供了对所有内部常量、论域和谓词的声明和实现。

这些内部常量、论域和谓词既可以用于编译单元（例如，在`#if ... constructions`中）也可以应用于实现（运行时间支持这些程序）。

每一编辑单元都隐含着嵌入式隐藏类的声明，但事实上这一类有着专门的惟一内部名，在代码的任何地方都不能直接引用。在内置项的名字前可以加“`::`”进行限定。

注意：子句变量 `This` 是在对象谓词子句中自动定义的。

下面的表 14.1、表 14.2、表 14.3 分别是内部常量、内部论域、内部谓词的简要描述。

表 14.1 内部常量简述

内部常量	说明
<code>compilation_date</code>	编译日期
<code>compilation_time</code>	编译时间
<code>compiler_buildDate</code>	编译器建立时间
<code>compiler_version</code>	编译器版本
<code>maxFloatDigits</code>	定义编译器支持的 <code>digits</code> 的最大值
<code>null</code>	空指针
<code>platform_bits</code>	定义一编译平台的数位容量
<code>platform_name</code>	定义目标平台名称

表 14.2 内部论域简述

内部论域	说明	内部论域	说明
<code>char</code>	宽字符	<code>unsigned</code>	无符号整数
<code>string</code>	以 0 终止的宽字符序列	<code>real</code>	浮点数
<code>symbol</code>	以 0 终止的宽字符序列	<code>pointer</code>	指向内存地址的 4 字节指针
<code>binary</code>	字节序列	<code>boolean</code>	布尔值
<code>integer</code>	有符号整数	<code>factdb</code>	内部数据库的命名描述

表 14.3 内部谓词简述

内部谓词	说明
<code>* //2</code>	乘法算术运算
<code>+ //2</code>	加法算术运算

续表

内部谓词	说明
-//2	减法算术运算
/ //2	除法算术运算
assert/1 procedure (i)	在匹配的内部事实数据库的底部插入指定事实
asserta/1 procedure (i)	在匹配的内部事实数据库的顶部插入事实
assertz/1 procedure (i)	在匹配的内部事实数据库的底部插入事实
bound/1 determ (i)	检查一指定变量是否绑定到某个值
class_Name : () -> ::string ClassName procedure ()	这一编译时间谓词返回字符串 ClassName , 表示当前接口或类的名称
convert//2 procedure (i,i)	有检查的项转换
digitsOf//1 procedure (i)	返回指定浮点数论域的精度
div//2	算术运算符, 返回一整数除法的商
errorExi t : (::unsigned ErrorNumber) erroneous (i)	用指定返回代码 ErrorNumber 执行一次运行错误并设置内部错误信息
fail : () failure ()	调用回溯
finally/2 determ (i,i)	finally 元谓词使应用程序保证清除代码 Final_Predicates 的执行, 即使当代码块 Do_Predicates 的执行被中断。 Final_Predicates 在 Do_Predicates 之后立刻执行, 即使 Do_Predicates 退出或失败
findall/3 procedure (i,i,o)	收集一个非确定性谓词返回的所有解的列表
free/1 determ (i)	检查一变量是否是自由的
hasDomain/2 determ (i,i) procedure (i,o)	检查变量 VariableName 是否有论域 domainName
lowerBound//1 procedure (i)	返回指定数字论域的低界
maxDigits//1 procedure (i)	检索与浮点指针论域 domainName 相应的基本论域的数字值 (精度)
mod//2	算术运算符, 返回整数除法的余数
not/1 determ (i)	对子目标的结果 (成功 / 失败) 求反
predicate_fullname : () -> ::string PredicateFullName procedure ()	这一编译时间谓词返回字符串 PredicateFullName , 它表示子句体中的 predicate_name 得到调用的谓词名字。返回的谓词名用一作用域加以限制
predicate_name : () -> ::string PredicateName procedure ()	这一编译时间谓词返回字符串 PredicateFullName , 它表示在其子句体中 predicate_name 得到调用的谓词名称
retract/1 nondeterm (i) nondeterm (o)	从被匹配的内部事实数据库中除去一匹配的事实
retractall/1 procedure (i)	从被匹配的内部事实数据库中除去所有匹配的事实
retractall/2 procedure (i,i)	从被指定的内部事实数据库 FactsSectionName 中除去所有匹配的事实
sizeBitsOf//1 procedure (i)	检索内存中被指定论域 DomainName 的实体占用的位数
sizeOf//1 procedure (i)	检索内存中被指定项占用的字节数
sizeOfDomain//1 procedure (i)	检索内存中被指定论域 DomainName 的实体占用的字节数

续表

内部谓词	说明
sourceFile_LineNo : () -> ::unsigned procedure ()	返回在编译器中处理的源文件的当前行号
sourceFile_Name : () -> ::string procedure ()	返回在编译器中处理的源文件的名称
sourceFile_TimeStamp : () -> ::string procedure ()	返回表示编译器处理的源文件的日期和时间的字符串
succeed/0	谓词 succeed/0 总是成功
toBinary//1 procedure (i)	将指定项转换为 binary 表示
toBoolean//1 procedure (i)	这一元谓词的用途是将一确定性调用（谓词或事实）转换为一返回布尔论域值的程序
toString//1 procedure (i)	将指定的项转换成字符串表示
toTerm//1 procedure (i)	将指定项 SrcTerm 的字符串 / 二进制表示转换成与返回值的 PrologTerm 变量论域相应的表示
trap/3 determ (i,o,i)	在设陷阱的谓词中捕获退出、中断和运行错误
tryConvert//2 determ (i,i)	检查输入项 InputTerm 是否能严格地转换成指定论域 returnDomain ，并返回转换后的项 ReturnTerm
uncheckedConvert//2 procedure (i,i)	论域的无检查的转换
upperBound//1 procedure (i)	返回指定数字论域的上界值

14.2 内部常量详解

本节以字母顺序，详细介绍 Visual Prolog 的内部常量。

::compilation_date

编译日期。这里 YYYY 表示年，MM 表示月数，DD 表示天数。

```
compilation_date : ::string = "YYYY-MM-DD".
```

::compilation_time

编译时间。这里 HH 表示小时，MM 表示分钟，SS 表示秒。

```
compilation_time : ::string = "HH-MM-SS".
```

::compiler_buildDate

编译器建立时间。

```
compiler_buildDate : ::string = "YYYY-MM-DD HH-MM-SS".
```

::compiler_version

编译器版本，该值决定编译器版本。

```
compiler_version = 6003.
```

::maxFloatDigits

定义编译器支持的 `digits` 的最大值。

```
maxFloatDigits = 19.
```

::null

空指针或默认的 `NULL` 指针。

```
null : ::pointer = uncheckedConvert(::pointer, 0).
```

::platform_bits

定义一编译平台的数位容量。

```
platform_bits = 32.
```

::platform_name

定义目标平台的名字。

```
platform_name : string = "Windows 32bits".
```

14.3 内部论域详解

本节按字母顺序，详细介绍 `Visual Prolog` 的内部论域。

::char

宽字符集。

```
char
```

该论域的值是 `unicode` 字符，采用双字对字符编码。

只有赋值和比较（按字典顺序的意义来说）操作应用该论域的值。字符的映射有如下语法：

```
char_image :  
    ' char_value '
```

```

char_value :
    letter
    digit
    graphical_symbol
    \ escape_seq
escape_seq :
    t
    n
    r
    \
    '
    "
    u HHHH

```

在上面的语法中，HHHH 对应于 4 个十六进制的数字。同样，反斜线符号和单引用号只能用一种转义序列表示。

::string

宽的 0 终止的宽字符序列。

```
string
```

一个字符串是一串 **unicode** 字符。它作为指向宽的 0 终止的宽字符序列的指针。只有赋值和比较（按字典顺序的意义来说）操作应用该论域的值。在源代码中，一个字符串文字可用双引号内的一串字符定义。

```

stringLiteral:
    stringLiteralPart-list

```

```

stringLiteralPart:
    @" anyCharacter-list-opt "
    " characterValue-list-opt "

```

一个字符串文字由一个或多个连接在一起的字符串文字部件（**stringLiteralPart**）组成。以@开头的字符串文字部件不使用转义序列，而开头没有@的字符串文字部件用以下转义序列：

```

\\    表示\
\t    表示制表符
\n    表示换行符
\r    表示回车
\'    表示单引号
\"    表示双引号

```

一个 u 后面跟着 4 个十六进制数表示与数值对应的 **unicode** 字符。

字符串中的双引号只能用转义序列表示（单引号既可以用转义序列，也可以用图形符号表示）。

::symbol

宽的 0 终止的宽字符序列。

symbol

与字符串相似，一个符号也是一个 **unicode** 字符序列。它是通过指向包含字符串的符号表的入口的指针而实现的。可应用于符号的操作与可应用于字符串的操作相同。

一个符号的映像用 `<string_literal>` 表示（任何用双引号括起来的字符）。

符号和字符串大部分是可互换的，但它们存储的方式不同。符号保存在一个查询表中，而它们的地址不是符号本身被存储以代表对象。这意味着符号可以快速匹配，而且如果一个符号在一个程序中反复出现，它们可以非常简洁地存储。字符串并不存在于查询表中。无论何时字符串需要匹配，**Visual Prolog** 都会逐字符检查。

::binary

N 个字节的序列。

binary

该论域的值用于保存二进制数据。一个二进制值作为一个指向字节序列的指针实现，该指针代表二进制项的内容。

二进制项的长度紧接在字节序列之前的双字中。实际上这个双字包含

```
TotalNumberOfBytesOccupiedByBinary = ByteLen + 4
```

这里，**ByteLen** 是二进制项的长度，4 是双字占用的字节数。

只有赋值和比较操作使用二进制论域的值。

以下比较两个二进制项：

（1）如果它们的长度大小不一，则认为长的那个比较大。

（2）否则，将它们视为无符号值，逐字节比较。当发现两个不同字节时比较结束，它们比较的结果就是二进制项比较的结果。若两个二进制项长度相同，所有的字节也相同，则认为它们相同。

二进制映射文本语法由二进制规则决定。

```
binary:
    $ [ byte_value-comma-sep-list-opt ]
byte_value :
    expression
```

每一表达式应在编译时间计算，其值应在 0~255 之间。

::integer

有符号整型数。

integer

该论域的值占据 4 个字节。算术运算(+, −, /, *)、比较、赋值、div、mod 操作可应用该论域的值。

整数允许的范围为从−2 147 483 648~2 147 483 647。

整数映像的语法由整数规则确定：

```
integer :  
    dec_number  
    0x hex_number  
    0o oct_number  
oct_number :  
    oct_digit-list  
oct_digit : one of  
    0 1 2 3 4 5 6 7  
dec_number :  
    dec_digit-list  
dec_digit : one of  
    oct_digit 8 9  
hex_number :  
    hex_digit-list  
hex_digit : one of  
    dec_digit a b c d e f A B C D E F
```

::unsigned

无符号整型数。

unsigned

该论域的值占 4 个字节。算术操作(+, −, /, *)、比较、赋值、div 以及 mod 操作可应用该论域的值。

无符号整数映像的语法与整数的一样。无符号整数不能使用负号，允许数的范围为从 0~4 294 967 295。

::real

浮点数。

real

该实数论域仅为用户方便而引入。所有的算术、比较和赋值操作都可应用该实数论域的值。

实数允许的范围为 $1 \times 10^{-307} \sim 1 \times 10^{+308}$ ，即 $1e-307 \sim 1e+308$ 。在必要时，整数论域的值自动转换为实数论域。

浮点数映像的语法由实数规则确定：

```

real :
    fraction exponent-opt
fraction :
    dec_number fractional_part-opt
fractional_part :
    .dec_number
exponent :
    exp add_operation-opt dec_number
exp :
    e
    E
add_operation :
    +
    -
dec_number :
    dec_digit-list
dec_digit : one of
    0 1 2 3 4 5 6 7 8 9

```

::pointer

指向内存地址的 4 字节指针。

```
pointer
```

一个指针直接与内存地址对应并由 4 字节值实现。只有等于操作可应用该论域的值。指针的映像不能在源文件中显式写出。只有内置的空常量 `null` 可获得该论域的空值 `NULL`。

::boolean

布尔值。

```
boolean
```

该论域仅为用户方便而引入。可以像一般具有下列定义的混合论域来对待它。

```

domains
    boolean = false(); true()

```

::factdb

命名内部数据库的描述符。

```
factdb
```

该论域有如下的隐藏声明：

```
domains
```

```
factdb = struct @factdb( named_internal_database_domain, ::object ).
```

所有用户定义的事实段的名称都是该论域中的常数。只要有必要，编译器会自动从这些常数中建立相应的混合项。运行时，这一结构的第 1 个字段包含相应论域描述符的地址，第 2 个字段包含着零（对于类事实段而言）或指向对象的指针 **This**（对于对象事实段而言）。

14.4 内部谓词详解

本节按字母顺序，详细介绍 Visual Prolog 的内部谓词。

::* //2

```
* //2.
```

乘法算术运算。

描述：

乘法既可应用于整数也可应用于浮点数。若操作数之一为浮点数，则结果也是浮点数。

::+ //2

```
+ //2.
```

加法算术运算。

描述：

加法既可应用于整数也可应用于浮点数。若操作数之一为浮点数，则结果也是浮点数。

::- //2

```
- //2.
```

减法算术运算。

描述:

减法既可应用于整数也可应用于浮点数。若操作数之一为浮点数,则结果也是浮点数。

:://2

```
/ //2.
```

除法算术运算。

描述:

除法既可应用于整数也可应用于浮点数,结果总是浮点数。

::assert/1

```
assert/1  
procedure (i).
```

在被匹配的内部事实数据库的底部插入指定事实。

描述:

assert(Fact) 谓词在被匹配的内部事实数据库插入一个事实,插入点位于为相应的数据库谓词存储的其他任何事实之后。该事实必须是一个属于内部事实数据库论域的项。用于单个事实的 **assert/1** 将已有事实的实例变为指定的事实。**assert/1** 与 **assertz/1** 的作用相同。

参见 **assertz/1**。

异常:

对声明为 **determ** 的事实进行声明,但该事实实例已经存在。

::asserta/1

```
asserta/1  
procedure (i).
```

在被匹配的内部事实数据库顶部插入事实。

描述:

asserta(Fact) 谓词在被匹配的内部事实数据库插入一个事实,插入点位于为相应谓词存储的其他任何事实之前。该事实必须是一个属于内部事实数据库论域的项。用于单个事实的 **asserta/1** 将已有事实的实例变为指定的事实。参见 **assert/1** 和 **assertz/1**。

异常:

对声明为 **determ** 的事实进行声明,但该事实实例已经存在。

::assertz/1

```
assertz/1  
procedure (i).
```

在匹配内部事实数据库底插入事实。

描述:

`assertz(Fact)` 谓词在被匹配的内部事实数据库插入一个事实，插入点位于为相应的数据库谓词存储的其他任何事实之后。该事实必须是一个属于内部事实数据库论域的项。用于单个事实的 `assertz/1` 将已有事实的实例变为指定的事实。参见 `assert/1` 和 `asserta/1`。

异常:

对声明为 `determ` 的事实进行声明，但该事实实例已经存在。

::bound/1

```
bound/1
determ (i).
```

检查一指定变量是否绑定到一个值上。

描述:

若变量已绑定，则 `bound (Variable)` 成功，若变量是自由的，则 `bound (Variable)` 失败。`bound` 用来控制流模式并检查引用变量的绑定。如果指定变量的任一部分已初始化，`bound` 谓词视其为已绑定。参见 `free/1`。

::class_Name/0

```
class_Name : ()
-> ::string ClassName
procedure ().
```

这一编译时间谓词返回字符串 `ClassName`，表示当前接口或类的名字。

::convert//2

```
convert//2
procedure (i,i).
```

有检查的项转换。

描述:

该函数的调用模板如下:

```
ReturnTerm = convert ( returnDomain, InputTerm )
```

参数:

`returnDomain`

指定一论域，`convert//2` 函数将 `InputTerm` 转换到该论域。这里，`returnDomain` 必须是一内置 Visual Prolog 论域名、接口论域或一用户定义的与一个内部 Visual Prolog 论域同义的论域名。论域名 `returnDomain` 必须在编译时间指定，即它不能由变量得来。

InputTerm

指定要转换的值。InputTerm 可以是任何 Prolog 项或表达式。若 InputTerm 为一表达式，那么它将在转换前得到计算。

ReturnTerm

返回参数 ReturnTerm 将是 returnDomain 类型。

注意：转换谓词对给定的 InputTerm 进行清除和真正的转换，返回一指定新论域 returnDomain 的项 ReturnTerm。如果不能进行要求的转换，就产生错误。tryConvert 谓词提供相似的功能，但在不能执行转换时，tryConvert 不产生任何执行错误。

允许的转换：

- 数字论域之间的转换；
- 接口类型之间的转换；
- string 和 symbol 论域之间的转换；
- 从 binary 到 pointer；
- 上述论域的同义字；
- 引用论域和相应的非引用论域之间。

与之相对应的是无检查的转换谓词 uncheckedConvert//2，它完成任意两个论域之间不加限制的转换，只要两论域之间具有相同的位长度。

当源论域和目标论域在编译过程中都静态已知时，convert//2 (或 tryConvert//2)谓词完成有检查的显式转换。显式转换的结果可以是如下之一：

- ok，成功地转换至目标论域；
- run-time-check，为兼容性产生的运行时间检查向目标论域的转换；
- error，不可能转换，错误输出。

有检查的显式转换规则：

- 论域的同义字按论域本身转换的规则转换。
- 数字论域只能转换为数字论域。
- 整型常量代表匿名整数论域：[const .. const]。
- 实型常量代表匿名浮点数论域：digits dig [const .. const]，这里 dig 是不包括无意义的零的尾数数位的个数。
- 符号论域的值可以转换为字符串论域，反之亦然。
- 二进制论域的一个值可以转换为指针论域。
- 为接口隐含引入的论域只可以按下述规则转换为接口论域。
- 所有其他的论域不能转换。

数字论域的转换：

- 这种转换中，范围是首先要考虑的。若源和目标的范围不相交，则产生错误。如果源和目标的范围部分相交，则产生一致性运行检查。同样，如果一个论域是实数而另一个是整数，则在比较前，整数论域转换为实数论域。
- 当输入项是实数而输出项是整数，则 convert//2 和 tryConvert//2 谓词截断输入值为靠近 0 的最邻近的整数值。

接口类型的转换:

谓词 `convert//2` 可将任何对象转换为接口类型。实际的正确性在运行中检查。当对象被创建时, 它的类型存储在内部, 因此当对象作为参数被传递时, 它仍然保留它最初类型。这一最初类型用来检查转换的允许与否。

举例:

```
interface x
    supports a, b
end interface x
```

如果对象是由实现 `x` 接口的类创建的, 则对象作为类型为 `a` 的参数传递给某些谓词, 然后它被允许转换为 `b` 类型的对象。

异常:

检查范围错误。

不支持的接口类型。

::digitsOf//1

```
digitsOf//1 -> unsigned
    procedure (i).
```

返回指定浮点论域的精度。

描述:

该函数的调用模板为

```
Precision = digitsof ( domainName )
```

该编译时间谓词的输入参数 `domainName` 为一浮点论域, 它应在编译时间显式指定(也就是说, `domainName` 不能由变量得到)。谓词返回的数字 `Precision` 由论域声明中的 `digits` 属性决定。

编译器保证论域 `domainName` 的值不少于有意义小数的 `Precision` 位数。

::div//2

```
div//2.
```

算术运算, 返回一整数除法的商。

描述:

操作数和结果都是整数, `I div K` 返回 `I` 被 `K` 除的商。

::errorExit/1

```
errorExit : (::unsigned ErrorNumber)
    erroneous (i).
```

执行具有指定返回代码 `ErrorNumber` 的运行时间错误，它可用在 `trap/3` 中。

::fail/0

```
fail : ()  
      failure ().
```

调用回溯。

描述：

`fail` 谓词强迫谓词失败，因此总是导致回溯。在一个以失败结束的子句中，它不能为子句绑定输出参数。

::finally/2

```
finally/2  
determ (i,i).
```

`finally` 元谓词使应用程序保证清除代码 `Final_Predicates` 的执行，即使当代码块 `Do_Predicates` 的执行被打断。`Final_Predicates` 在 `Do_Predicates` 之后立刻执行，即使 `Do_Predicates` 退出或失败。

描述：

该谓词调用模板如下：

```
finally ( Do_Predicates, Final_Predicates )
```

该元谓词对强制动作的安全程序设计实现有帮助，就像在执行正确或不正确的操作后保证释放资源一样。

`Do_Predicates` 和 `Final_Predicates` 都必须是 `determ`（或更强）。从 `Do_Predicates` 的谓词输出的变量（如果有）不能用在 `Final_Predicates` 的谓词中。

谓词 `finally/2` 有以下语义：

首先执行 `Do_Predicates`。

如果它们失败或者发生任何意外，执行 `Final_Predicates`。所有在 `Do_Predicates` 的执行过程中获得值的变量在失败或意外发生时变得不确定。因此任何使用它们的企图（在 `Final_Predicates` 中或谓词 `finally` 外）都是错误的。

如果 `Do_Predicates` 顺利执行，则 `Final_Predicates` 在 `Do_Predicates` 的最后一个语句被执行后，立即正常执行。

`finally` 结束时产生以下结果：

如果 `Do_Predicates` 失败或产生意外，则 `finally` 以失败或相应意外而终止（如果在 `Final_Predicates` 中没有失败或意外时）。若 `Do_Predicates` 成功，则 `finally` 根据 `Final_Predicates` 的结果终结。若 `Do_Predicates` 产生意外 `N1`，接着 `Final_Predicates` 产生意外 `N2`，那么 `finally` 以意外 `N2` 终结。

::findall/3

```
findall/3  
procedure (i,i,o).
```

搜集不确定性谓词返回的所有解的列表。

描述:

该谓词的调用模板为

```
findall ( ArgumentName, predicateCall, ValuesList )
```

参数:

ArgumentName

在指定谓词 **predicateCall** 的参数中, 指定哪个参数将被搜集进表 **ValuesList**。

predicateCall

指出从中搜集值的谓词。

ValuesList

输出变量, 保存通过回溯搜集到的值的列表。

注意:谓词 **findall** 通过回溯至谓词 **predicateCall** 来建立表 **ValuesList**, 直至它失败。**ValuesList** 由变量 **ArgumentName** 的所有实例组成, 该变量必是指定谓词 **predicateCall** 的一个输出参数。

为使 **findall** 工作, 必须在用户论域声明中对列表论域进行声明。这里不允许自由变量存在。

::free/1

```
free/1  
determ (i).
```

检查变量是否是自由的。

描述:

该谓词调用模式为

```
free ( Variable )
```

若指定的变量是自由的, 则谓词 **free** 成功; 若指定的变量已被绑定, 则谓词 **free** 失败。如果指定变量的任一部分被绑定, 谓词 **free** 视其为已绑定。参见 **bound/1**。

::hasDomain/2

```
hasDomain/2  
procedure (i,o).
```

检查变量 `VariableName` 是否有论域 `domainName`。

描述：

该谓词调用模式为

```
hasDomain ( domainName, VariableName ) (i, o)
```

该谓词创建属于由参数 `domainName` 定义的论域的自由变量 `VariableName`。参数 `domainName` 必须是标准的或用户定义的论域；该论域名必须在编译时间显式指定（即它不能由变量得来）。

::lowerBound//1

```
lowerBound//1
  procedure (i).
```

返回指定数字论域的低界。

描述：

该谓词调用模式为

```
LowerBoundValue = lowerBound ( domainName )
```

`LowerBound` 是一个编译时间谓词。`lowerBound` 返回指定数字论域 `domainName` 的低界绑定值。返回值 `LowerBoundValue` 属于与 `domainName` 相同的论域。参数 `domainName` 应是任一数字论域的名字；该论域名必须在编译时间显式指定（即它不能由变量得来）。参见 `upperBound//1`。

异常：

如果指定论域 `domainName` 不是数字论域，则发生编译时间错误。

::maxDigits//1

```
maxDigits//1 -> unsigned
  Sprocedure (i).
```

检索与浮点指针论域 `domainName` 相应的基本论域的数字值（精度）。

描述：

该谓词调用模式为

```
MaxDigitsNumber = maxdigits ( domainName )
```

对 Visual Prolog 6 而言，返回值 `MaxDigitsNumber` 总是等于 19。参数 `domainName` 应是任一浮点论域的名称；该论域名必须在编译时间显式指定（即它不能由变量得来）。返回值论域为无符号数。

异常：

如果指定论域 `domainName` 不是浮点论域，则发生错误。

::mod//2

```
mod//2.
```

算术运算，返回整数除法的余数。

描述：

操作数和结果都是整数， $I \bmod K$ 返回 I 被 K 除的余数。

::not/1

```
not/1
determ (i).
```

对子目标的结果（成功 / 失败）求反。

描述：

该谓词调用模式为

```
not ( PredicateCall )
```

若 `PredicateCall` 代表的目标在求值时失败，则 `not` 成功。在对 `not` 的调用中，命名的输出变量是不允许使用的，因为变量不能在 `not` 操作中进行绑定。将 `not` 视为“若条件真则失败”或“若条件假则成功”是个好主意。如果调用 `PredicateCall` 成功则方法失败。

::predicate_Fullname//0

```
predicate_FullName : ()
-> ::string PredicateFullName
procedure ().
```

这一编译时间谓词返回一个字符串 `PredicateFullName`，它表示在其子句体中 `predicate_name` 得到调用的谓词名字。返回的谓词名用一作用域名进行限制。

描述：

`predicate_FullName` 仅可用在谓词定义的子句体内。在其他地方使用 `predicate_FullName` 将导致编译时间错误。参见 `predicate_Name`。

::predicate_name//0

```
predicate_name : ()
-> ::string PredicateName
procedure ().
```

这一编译时间谓词返回字符串 `PredicateFullName`, 它表示在其子句体中 `predicate_name` 得到调用的谓词名称。

描述:

`predicate_Name` 仅可用在谓词定义的子句体内。在其他地方使用 `predicate_Name` 将导致编译时间错误。参见 `predicate_FullName`。

::retract/1

```
retract/1
  nondeterm (i)
  nondeterm (o).
```

从被匹配的內部事实数据库中除去一匹配的事实。

描述:

该谓词调用模式为

```
retract( FactTemplate )
```

这里事实模板 `FactTemplate` 是一事实项。谓词 `retract/1` 在合适的事实数据库中删除第 1 个与 `FactTemplate` 匹配的事实。其他匹配事实将在回溯中删除。

注意: `FactTemplate` 可以有任何级别的实例。`FactTemplate` 与事实数据库中的事实匹配, 这意味着任何自由变量在 `retract/1` 的调用中都将绑定。

`FactTemplate` 可以包含任何匿名变量, 也就是说, 如果变量仅在子句体中出现一次, 那么变量名可以是单个下划线或以下划线开头的变量名。例如:

```
retract ( person("Hans", _Age) )
```

将撤销第 1 个匹配的 `person` 事实, 该事实第 1 个参数为 "Hans", 第 2 个参数任意。

当撤销一个声明为 `determ` 的事实时, `retract/1` 的调用将是确定性的。参见 `retractall/1` 和 `retractall/2`。

警告:

如果在项目的当前范围内声明了单个事实, 那么应注意在调用 `retract/1` 时要具有自由 `FactTemplate` 变量。如果撤销单个事实, 就会发生运行时间错误。当没有更多的匹配时, 方法失败。

::retractall/1

```
retractall/1
  procedure (i).
```

从被匹配的內部事实数据库中删除所有匹配的事实。

描述:

该谓词调用模式为

```
retractall ( FactTemplate )
```

这里 **FactTemplate** 应是一事实项。

retractall/1 谓词为事实库中所有的事实数据库谓词撤销所有与 **FactTemplate** 匹配的谓词。它总是成功，即使无事实可撤销。

从 **retractall/1** 不可能得到任何输出值。因此，调用中的变量必须被绑定或是单个下划线(匿名)。注意，**FactTemplate** 可以有任何级别的实例，但自由变量必须是单下划线(“无条件匿名”)。与 **retract/1** 不同，以下划线开头的“有条件匿名”变量(比如 **_AnyValue**) 在 **retractall/1** 中不能使用。参见 **retract/1** 和 **retractall/2**。

::retractall/2

```
retractall/2
  procedure (i,i).
```

从指定的内部事实数据库 **FactsSectionName** 中删除所有匹配的事实。

描述:

该谓词调用模式为

```
retractall ( FactTemplate, FactsSectionName )
```

retractall/2 从指定的事实数据库 **FactsSectionName** 中删除所有与 **FactTemplate** 匹配的事实。

从 **retractall/2** 不可能得到任何输出值。因此，调用中的变量必须被绑定或是单个下划线(匿名)。

名为 **FactsSectionName** 的事实数据库应在编译时指定，它不能从变量中获得。

retractall/2 是确定性的，并将总是成功。

参见 **retractall/1** 和 **retract/1**。

::sizeBitsOf//1

```
sizeBitsOf//1 -> unsigned
  procedure (i).
```

检索内存中被指定论域 **DomainName** 实体占用的位数。

描述:

该谓词调用模式为

```
Bits_Size = sizeBitsOf ( DomainName )
```

这一编译时间谓词接收论域 **DomainName** 作为输入参数，并返回给定论域占用内存的

位数。结果度量单位为位。对实数论域而言，谓词 `sizeBitsOf/1` 返回论域声明中长度字段所定义的值。

对整数论域而言，以下总是正确的。

```
sizeofDomain(domain)*8 - 7 <= sizeBitsOf(domain) <= sizeofDomain(domain)*8
```

参见 `sizeofDomain/1`。

::sizeOf/1

```
sizeof//1 -> unsigned
procedure (i).
```

检索内存中被指定项占用的字节数。

描述：

该谓词调用模式为

```
ByteSize = sizeof ( Term )
```

函数 `sizeof/1` 接收一个项(`Term`)作为输入参数，并返回无符号值字节长度(`ByteSize`)，该值代表了内存中该项 (`Term`) 占用的字节数。

::sizeofDomain/1

```
sizeofDomain//1 -> unsigned
procedure (i).
```

检索内存中被指定论域 `DomainName` 实体占用的字节数。

描述：

该谓词调用模式为

```
Bytes_Size = sizeofDomain ( DomainName )
```

这一编译时间谓词接收论域 `DomainName` 作为输入参数，并返回被给定论域占用内存的大小。结果度量单位为字节。返回值 `Bytes_Size` 属于无符号数。与 `sizeBitsOf/1` 相比，`sizeBitsOf/1` 返回用位数度量的项的大小，而该谓词返回用字节数度量的论域的大小。

::sourceFile_LineNo//0

```
sourceFile_LineNo : ()
-> ::unsigned
procedure ().
```

返回在编译器中处理的源文件的当前行号。

描述:

这一编译时间谓词返回正在编译的文件中正由编译器处理的行号。

::sourceFile_Name//0

```
sourceFile_Name : ()  
-> ::string  
procedure ().
```

返回在编译器中处理的源文件的名称。

描述:

这一编译时间谓词返回正在编译的文件名称的字符串。

::sourceFile_TimeStamp//0

```
sourceFile_TimeStamp : ()  
-> ::string  
procedure ().
```

返回表示在编译器中处理的源文件的日期和时间的字符串。

描述:

这一编译时间谓词返回当前被编译源文件的时间和日期的字符串，字符串格式为 YYYY-MM-DD HH:MM:SS 的形式。这里，

YYYY——年

MM——月

DD——日

HH——小时

MM——分钟

SS——秒

::succeed/0

```
succeed/0.
```

谓词 succeed/0 总是成功。

描述:

标准谓词 succeed/0 正好成功一次。

::toBinary//1

```
toBinary//1 -> binary  
procedure (i).
```

将指定项转换为二进制表示。

描述：

该函数调用模式为

```
Binary_Term = toBinary ( Term )
```

当 **Term** (属于某论域 **domainName**) 被转换为二进制时，它可安全地存储于一文件内或通过网络送到另一程序中。以后，通过函数 **toTerm//1** 得到的二进制值 **Binary_Term** 可以转换回 **Visual Prolog** 项（被转换项的论域必须适合 **domainName**）。

::toBoolean//1

```
toBoolean//1 -> boolean  
procedure (i).
```

这一元谓词的用途是将一确定性调用（谓词或事实）转换为返回值为 **boolean** 论域的程序。

描述：

该谓词调用模式为

```
True_or_False = toBoolean ( deterministic_call )
```

toBoolean//1 元谓词返回 **boolean** 值。如果 **deterministic_call** 成功，则结果为真；如果 **deterministic_call** 失败，则结果为假。

::toString//1

```
toString//1 -> string  
procedure (i).
```

将一指定的项转换成字符串表示。

描述：

该谓词调用模式为

```
String_Term = toString ( Term )
```

当 **Term** (属于某论域 **domainName**) 被转换为字符串时，它可安全地存储于一文件内或通过网络送到另一程序中。以后，通过函数 **toTerm//1** 得到的字符串可以转换回 **Visual Prolog** 项（被转换项的论域必须适合 **domainName**）。

::toTerm//1

```
toTerm//1  
procedure (i).
```


将指定项 `SrcTerm` 的字符或二进制表示转换成与返回值的 `PrologTerm` 变量的论域相应的表示。

描述:

该谓词调用模式为

```
PrologTerm = toTerm ( SrcTerm )
```

编译时, 编译器应能确定返回值 `PrologTerm` 的论域。注意, 谓词 `toTerm` 的二进制形式执行几乎是逐字节的转换, 并只检查 `SrcTerm` 数据与返回值 `PrologTerm` 要求的论域的一般兼容性。程序员完全负责提供能够被正确转换为期望论域的项的 `SrcTerm` 二进制数据。谓词 `toTerm` 与谓词 `toBinary//1` 和 `toString//1` 相似。当 `Term` (属于某论域) 被转换为二进制或字符串表示 `SrcTerm` 时 (分别通过 `toBinary//1` 和 `toString//1`), 它可安全地存储于一文件内或通过网络送到另一程序中。以后, 通过相应的函数 `toTerm//1`, 将得到的字符串或二进制值 `SrcTerm` 转换回 Visual Prolog 项 `PrologTerm`。为保证反转换的正确性, 子句变量 `PrologTerm` 的论域应适合初始论域 `domainName`。

异常:

若编译器不能确定返回值的论域, 则发生编译时间错误。

当谓词 `toTerm//1` 不能将字符串或二进制值转换为指定论域的项时, 产生运行错误。

::trap/3

```
trap/3
determ (i,o,i).
```

在谓词 `PredicateCall` 中捕获退出、中断和运行时间错误。

描述:

该函数调用模式为

```
trap ( PredicateCall, ErrorCode, ErrorHandler )
```

谓词 `trap` 执行错误捕获和异常处理。通常, 如果 `PredicateCall` 成功, `trap` 就会完全成功, `PredicateCall` 失败, `trap` 也失败。然而, 如果在 `PredicateCall` 执行的过程中发生异常 (或在其子目标执行的过程中发生异常), 则 `ErrorCode` 将接收合适的错误代码值, 而 `ErrorHandler` 将得到调用。当 `ErrorHandler` 返回时, `trap` 也将返回。

若 `PredicateCall` 是不确定的, 这将在 `trap` 调用中反映出来。这意味着捕获一个子目标是完全透明的, 除非子目标退出。

在 `trap` 调用 `ErrorHandler` 之前, 堆栈、堆和指针将设置返回 `trap` 调用前它们所具有的值。这意味着 `trap` 能捕捉内存溢出。

如果 `PredicateCall` 失败或在调用错误处理器 `ErrorHandler` 后出现异常, 则该方法 (`Method`) 失败。

注意: 编译器暂时有以下限制, 即 `ErrorHandler` 谓词必须用错误或失败模式声明。

::tryConvert//2

```
tryConvert//2
    determ (i,i).
```

检查输入项 (InputTerm) 是否能严格地转换成指定论域 returnDomain 并返回转换后的项 (ReturnTerm)。

描述:

该函数调用模式为

```
ReturnTerm = tryConvert ( returnDomain, InputTerm )
```

参数:

returnDomain

指定一个论域, 谓词 tryconvert//2 试图转换指定的 InputTerm 至该论域中。这里, returnDomain 可以是当前作用范围内可得到的任意项。名为 returnDomain 的论域必须在编译时间指定, 即它不能从变量中得到。

InputTerm

指定必须转换的项。InputTerm 可以是任何 Prolog 项或表达式。如果 InputTerm 是一表达式, 那么它将在转换前得到计算。

ReturnTerm

返回项 ReturnTerm 将为 returnDomain 论域。

注意: 转换规则与嵌入谓词 convert//2 一样, 但是当 convert//2 产生转换错误时, tryconvert//2 失败。

若相应转换成功则该谓词成功, 否则失败。谓词 tryconvert//2 试图执行一次清除操作, 并且执行从已知 InputTerm 到指定论域 returnDomain 的值的真实转换。如果要求的转换不能执行, 谓词 tryconvert//2 失败。当谓词 tryconvert//2 成功时, 它返回转换至指定论域 returnDomain 的项 ReturnTerm。

关于有检查的显式转换, 其允许的转换和规则参见谓词 convert//2。

参见 uncheckedConvert//2。

::uncheckedConvert//2

```
uncheckedConvert//2
    procedure (i,i).
```

无检查的论域转换。

描述:

该函数调用模式为

```
ReturnTerm = uncheckedConvert ( returnDomain, InputTerm )
```

参数:

returnDomain

指定一个论域，谓词 `uncheckedConvert` 不安全地转换指定的输入项 (`InputTerm`) 到该论域中。这里返回论域 `returnDomain` 可以是当前作用域内可得到的任意项。返回项 (`ReturnTerm`) 与输入项 (`InputTerm`) 大小应一样。名为 `returnDomain` 的论域必须在编译时间指定，即它不能从变量中得到。

InputTerm

指定必须转换的值。`InputTerm` 可以是任何 Prolog 项或表达式。如果 `InputTerm` 是表达式，那么它将在转换前得到计算。

ReturnTerm

返回的参数 `ReturnTerm` 将为 `returnDomain` 类型。

注意: 谓词 `uncheckedConvert` 几乎执行任意的转换。谓词 `uncheckedConvert` 执行 `InputTerm` 的初步计算 (如果它是一表达式)，将现有的类型换成 `returnDomain` 类型并与 `ReturnTerm` 合一。谓词 `uncheckedConvert` 不执行运行时间的检查。它仅进行等同于被转换论域位长度的编译时间检查。所以几乎任何项都可以不顾一切地转换为其他任何项。所以如果试图使用被 `uncheckedConvert` 不正确地转换的变量，可能发生十分危险的结果。使用 `uncheckedConvert` 要格外小心，强烈建议在大多数情况下或必要时尽量使用 `convert//2` 和 `tryconvert//2`。当一个对象由 COM 系统返回时，有必要用 `uncheckedConvert` 来转换，因为 Prolog 程序没有它实际类型的信息。

::upperBound//1

```
upperBound//1
procedure (i).
```

返回指定数字论域的上界值。

描述:

该函数调用模式为

```
UpperBoundValue = upperBound ( domainName )
```

`upperBound` 是一编译时间谓词。`upperBound` 返回指定数字论域 `domainName` 的上界值。返回值 `UpperBoundValue` 属于相同的论域 `domainName`。参数 `domainName` 应是任何数字论域的名字；该论域名应在编译时间显式指定 (也就是说，`domainName` 不能从变量中得到)。参见 `lowerBound//1`。

异常:

如果指定的论域 `domainName` 不是数字论域，则产生编译时间错误。

本章小结

本章介绍所有内部常量、论域和谓词的声明和描述。

习题 14

1. 内部论域、谓词和常量与自定义论域、谓词和常量在概念上有何异同？
2. 内部论域、谓词和常量与自定义论域、谓词和常量在使用上有何差别？

第 15 章 与其他编程语言接口

本章介绍 Visual Prolog 与其他编程语言的接口，帮助读者学会如何在 Visual Prolog 中调用 Win32 API，打开更广阔的编程世界的大门。

15.1 外部代码

所谓“外部代码”是指用其他编程语言（而不是用 Visual Prolog）所编写的代码。

Visual Prolog 能直接调用其他语言代码。本章解释这些概念和细节。直接调用外部代码是一种二进制级的底层调用，而非高级语言层面的高级调用。这在简单的例程中相当简单，但也可能出奇地复杂。可以肯定的一点：处理复杂调用需要非常熟悉 Visual Prolog 和其他编程语言。但是不要担心，实际上，在许多例程中所需要的交互是相当简单的。

15.2 关键问题

其他语言编译器和 Visual Prolog 编译器有很大的不同，这是由于它们是由不同的人所制作的，而且它们必须支持不同的语言特性。Visual Prolog 不可能和所有的外部语言代码交互，因为它不可能知道其他编译器所采用的规则。所以，要实现 Visual Prolog 和其他语言代码的交互，就要求这些代码必须遵循规定的方式。

为了调用外部代码（异种语言代码），必须访问这些代码。本章要处理的代码是直接链接到程序里，或者位于一个动态链接库（DLL）中。

首先，必须查找这些代码的位置。这通过一个名字来实现。如果代码是直接链接到程序里，就必须使用链接名；如果代码位于一个 DLL，则可以使用导出名。无论是用链接名还是用导出名，对 Visual Prolog 来说没有区别，但是当试图在外部代码（或系统）中寻找这个名字时，就有了区别。这里提到系统，是因为有时必须使用的名字在代码中根本不存在。因此，对于这个概念，仅使用链接名。

其次，在已经确定了外部代码的位置之后，接着必须传递输入参数并调用代码，代码被执行后，还必须获取其输出等。有许多不同的途径去完成此过程。显然，调用者和被调用者必须在这一点达成一致，即双方必须有调用约定。

第三，关于数据表示。不仅调用双方必须在参数等的传递上相一致，更重要的是双方要以相同方式解释所传递的字节。也就是说，双方的数据表示必须相同。

最后要注意的一点是内存管理。调用者和被调用者必须明确在必要时由哪方分配和释放内存。如果释放内存和分配内存不是由同一方进行，那么释放内存一方必须以正确的方式释放内存。

总之，调用外部代码有 4 个关键问题需要考虑：

- 链接名
- 调用约定
- 数据表示
- 存储管理

15.3 调用约定和链接名

这里将这两个问题放在一起讨论，是因为传统编译器使用的一些调用约定和链接名方案相关联。

链接名（或导出名）用于识别想要调用的代码。不同的编译器用不同的默认链接名，许多编译器有多种指定链接名的途径。在 Visual Prolog 中，可以用保留字 `as` 为一个谓词声明一个链接名。如：

```
predicates
  pppp:(integer)as"LinkName".
```

在 Visual Prolog 程序中，上面的谓词被命名为 `pppp`，但是它的链接名是 `LinkName`。

注意：仅类谓词有一个链接名，这意味着它必须在一个类中进行声明或在一个类实现的类谓词（`class predicates`）段进行声明。

Visual Prolog 支持大量不同的调用约定。这些约定也在谓词中声明，但用 `language` 保留字。

```
predicates
  qqqq:(integer)language c.
```

按照 C 编译器的调用约定，编译器在 C 程序中的名字前添加一个下划线来创建链接名。如果用 C 调用约定但不提供链接名，Visual Prolog 也将使用这个约定。注意：一直到 build 6107 编译器实际上使用的是另一种命名策略，也就是说必须用 `as` 来获取链接名。上例中 `qqqq` 的链接名是 `_qqqq`。如果声明一个链接名，就必须严格遵守这个规则：

```
predicates
  rrrr:(integer)language c as "LinkName".
```

`rrrr` 将用这个 C 调用约定，并有一个链接名 `LinkName`（即没有下划线）。

C++编译器通常用的是 C 调用约定，但是它们不依赖 C 链接名，因为 C++允许重载。也就是说，在 C++中相同的名字只要变量数目不同或者类型不同，就可以看作不同的函数使用。这些不同变量必须有不同的链接名。因此 C++编译器具有完善的命名机制，这种命名基于 C++名字、变量的数目和类型。这个过程可称为“命名熨烫（`name mangling`）”。

不同的编译器采用不同的命名熨烫算法，从而互不兼容。通常情况下，在被作为外部

代码访问的 C++ 程序中要使用明确的链接名，或者在一个输出 C 部分封装这个声明，使编译器使用 C 命名约定。

Visual Prolog 也支持 stdcall 调用约定。Microsoft Visual Basic 使用 stdcall 作为 Microsoft Win32 平台 API 调用的调用约定，许多 Pascal 家族编译器，包括 Borland Delphi 也使用 stdcall。实际上，C 和 C++ 程序和其他语言的程序接口时，通常也使用 stdcall 调用约定。Visual Prolog 对 stdcall 调用约定使用与 C 调用相同的命名约定。注意：一直到 build 6107 编译器实际上使用的是另一种命名策略，因此必须用 as 来获取链接名。即在 Prolog 名字前面加前缀下划线来创建链接名，但是如果指定了一个链接名，它将被严格使用。

Microsoft Win32 API 使用特殊的 stdcall 调用约定，它较 C 调用约定多了一些名字修饰。可参阅 Visual Prolog 语言参考手册。Visual Prolog 有一个专用的调用约定 apicall 来支持这个特殊的 stdcall。apicall 和 stdcall 仅在名字修饰上有不同之处。有了 apicall，用 as 声明的外部链接名也要被修饰。若需要一个有不同修饰的名字，就必须用 stdcall 并自己指定修饰名。

15.4 数据表示

数据表示有许多细节问题，已超出本章讨论范围。这里仅就 Visual Prolog 如何表示各种数据做简要叙述。

所输入的数字参数按其值进行传递，即数值直接压入调用堆栈。输出数字参数通过引用传递，即结果指针被压入调用堆栈。调用堆栈中整型数占 32 位，实型数占 64 位。

字符也可以用数字表示。

其他数据用一个指针来指向。通过直接将指针压入调用堆栈来传递输入参数。通过压入结果指针的指针（即指向指针的指针）来传递输出参数。

一个算符的论域由指向一块内存的指针来表示，这块内存首先保存了算符，随后是每个子部件。这些子部件直接由一个数字或指向真实数据的指针来表示。

如果一个算符的论域只有一种选择，这种情况下通常具有相同的值，所以跳过这个算符。注意：在 Visual Prolog 5 中，除非该论域用 struct 关键字声明，否则该算符出现；在 Visual Prolog 6 中，该算符永远不会出现。

用 align 限定符可以使算符表示有所不同，参阅语言参考的相关部分。

字符串由一个指针表示，它指向由零值空字符终止的字符序列，与 C 语言一样。

二进制论域由一个指针指向其二进制数据。其值等于数据长度加 unsigned 类型的大小，这个值紧接着存储在数据之前。

15.4.1 举例

假设想在 Visual Prolog 程序中调用一个 C 例程。在 C 中的声明如下：

```
int myRoutine(
```

```
wchar_t * TheString,
int BufferLength,
wchar_t * TheResult );
```

wchar_t*是 C 语言中的 unicode 字符串。这个函数有 3 个变量：

- TheString 是一个字符串
- Bufferlength 是一个整型数
- TheResult 是一个字符串

该函数返回一个整型值。相应的 Visual Prolog 6 声明如下（假定它被声明在一个类中）：

```
class predicates
  myRoutine : (
    string TheString,
    integer BufferLength,
    string TheResult)
  -> integer
  language c.
```

若谓词在类中声明，则 class 应去掉。

15.4.2 外部链接库

如果声明的谓词如上例所示，编译器就提示一个错误：找不到声明谓词的字句。这并不奇怪，因为这个谓词根本不能在 Visual Prolog 中实现。这时就必须通知编译器，这个谓词的代码在外部（externally）某处。这里之所以使用 externally 这个单词，是因为它完全可以准确地表达声明该谓词的类的实现中用一个所谓的求解限定符所要表达的意思。如果这个类命名为 xxx，则它应该是

```
implement xxx
  resolve
    myRoutine externally
  ...
```

这样编译器就接受了这个声明，但还可能出现链接错误：_myRoutine 没有定义。这是因为，包含_myRoutine 的库还没有链接。

如果_myRoutine 位于一个静态库（一个 LIB 文件），则要把这个文件加入项目，链接器就会提取相关代码并加入程序。

如果_myRoutine 位于一个 DLL 库，仍有一个 LIB 文件用以描述这个 DLL，接着就把这个文件加入项目。在这种情况下，链接器在程序中放置信息，这些信息告知在哪里可以找到 DLL 例程。

如果_myRoutine 位于一个 DLL 库，但是没有对应的 LIB 文件，可以通过下面方式调用该例程：

```
implement xxx
```



```
resolve
    myRoutine externally from "myDLL.dll"
...
```

这样，当 `myRoutine` 被首次调用时，编译器就增加代码，动态加载 DLL 库。这个 DLL 库在程序退出之前不卸载。

`pfc/application/useDll` 可以用于动态加载和卸载 DLL 库。

15.5 存储管理

数字和字符参数被直接压栈或写入内存，这种情况比较简单，无需再详细叙述。

然而，不直接压栈的数据涉及许多内存管理问题。因为只要调用双方的任一方要访问这个数据，数据都必须存在于内存。一旦数据不再需要，就必须回收内存以防止内存漏掉。

通常情况下，谁分配内存谁才能去回收这些内存，因为另一方并不知道如何回收。当然可以让双方采用相同的机制。例如，一方可以让另一方看到它的机制（如作为导出例程）。

Visual Prolog 6 使用的内存分配例程是采用运行时间系统（`Vip6kernel.dll`）来实现的，并且可以从外部代码进行调用。

程序和库代码在处理何时回收内存时，通常采用一些事先已约定的方法。

15.5.1 典型解决方案

这一节介绍解决内存管理问题的通用方法。只要不涉及 COM 和 .NET，这通常认为是最通用的方法。原理很简单：

- 输入参数仅在调用期间存活，所以，如果被调用者需要存储某一输入参数以备后用，它就必须将这个参数复制到自己的内存区。
- 输出被复制到调用者的内存缓冲区，所以被调用者分配的内存不会传送给调用者。

前面的 `myRoutine` 就是一个典型的例子。输出写入 `TheResult`（由调用者分配的一个字符串），这就是为什么它看起来既是例程的输入又要传递 `BufferLength` 的原因。

Visual Prolog 6 程序调用 `myRoutine` 例程：

```
clauses
    p(TheString) = TheResult :-
        TheResult = string::create(bufferLength),
        RetCode = myRoutine(TheString, bufferLength, TheResult),
        checkRetCode(RetCode).
```

`sring::create` 为字符串分配内存，`myRoutine` 将它的结果复制到 `TheResult`（Visual Prolog 程序不涉及这些），它不回收任何已分配的内存。

15.5.2 垃圾收集和全局堆栈

如果不采用上面的典型方案，Visual Prolog 6 还提供一种全局性的 G-堆栈，并且用一个垃圾收集箱来管理堆。

关于 G-堆栈，这里建议：如果不采用典型解决方案，就不要迁移 G-堆栈存储器。如果不能确定数据是否在 G-堆栈，那么可以用谓词 `memory::toPersistentStorage` 来获得一个副本，以确保不处于 G-堆栈之中。

应该注意到，垃圾收集箱是一种循环回收机制：当某块内存不再需要时，不应该立即回收，而要查看在外部代码中是否还要访问它。通常这都会自动处理，不存在什么问题。但是垃圾收集箱不知道（也不必知道）数据在外部代码中何时存在。因此，数据在 Visual Prolog 程序中存在的时间要长于在外部代码中存在的时间。

一种典型的保留内存的方法是将它插入事实段，然后在外部代码不再需要它时回收它。

15.6 Win32 API 函数

也许程序从来不会调用其他编程语言的代码，因此可能会认为学习这一部分没有必要。但要知道整个操作系统对于程序来说都是外部的，因此学习 Win32 API 很有必要。

操作系统提供成千上万有趣的外部例程，可以利用本章的原则来调用它们。这些例程称为 Microsoft Windows XXX API 函数。不同平台提供的 API 有所不同，例如，Windows XP 比 WindowsNT 4.0 提供更多的例程。

各平台 API 的在线文档可参见 MSDN 库。

从 Prolog 的观点来看，这个文档有一个问题：它用到许多字符常量，但却没有定义这些常量的值。例如，例程 `PlaySound` 的文档说明可以用标志参数 `SND_ASYNC` 在程序中异步播放声音（即在应用程序继续执行期间播放声音）。`SND_ASYNC` 实际上是一个常数，但文档中并没有说明这个常数的值。

从平台 SDK 更新处下载 SDK C/C++，在其中的*.h 头文件中可以找到这些常量的宏定义。

至此，应该能编写类似下面的代码：

```
implement playSoundFile
  open core
  resolve playSound externally

  domains
    soundFlag = unsigned32.

  class predicates
    playSound : (string Sound,
```

```

pointer HModule,
soundFlag SoundFlag)
-> booleanInt Success
language apicall.

```

constants

```

snd_sync : soundFlag = 0x0000.
/* play synchronously (default) */
snd_async : soundFlag = 0x0001.
/* play asynchronously */
snd_nodetault : soundFlag = 0x0002.
/* silence (!default) if sound not found */
snd_memory : soundFlag = 0x0004.
/* Sound points to a memory file */
snd_loop : soundFlag = 0x0008.
/* loop the sound until next sndplaysound */
snd_nostop : soundFlag = 0x0010.
/* don't stop any currently playing sound */
snd_nowait : soundFlag = 0x00002000.
/* don't wait if the driver is busy */
snd_alias : soundFlag = 0x00010000.
/* name is a registry alias */
snd_alias_id : soundFlag = 0x00110000.
/* alias is a predefined id */
snd_filename : soundFlag = 0x00020000.
/* name is file name */
snd_resource : soundFlag = 0x00040004.
/* name is resource name or atom */
snd_purge : soundFlag = 0x0040.
/* purge non-static events for task */
snd_application : soundFlag = 0x0080.
/* look for application specific association */

```

clauses

```

run() :-
    console::init(),
    _ = playSound("tada.wav",
        null, snd_nodetault+snd_filename).

```

end implement playSoundFile

goal

```

mainExe::run(playSoundFile::run).

```

本章小结

对于现有主流平台上的任何一种编程语言，应该能够与其他编程语言无缝地接口，以实现混合语言编程功能。这是现代编程语言的一种共同特色。

本章介绍 Visual Prolog 与其他编程语言的接口，旨在帮助读者学会如何在 Visual Prolog 程序中调用 Win32 API，以解决 Visual Prolog 语言与其他编程语言之间的接口或混合语言编程问题。

习题 15

1. 从 Visual Prolog 调用其他编程语言时，必须理解哪些方面的问题？
2. 何谓“外部代码”？从 Visual Prolog 调用“外部代码”需要解决哪些关键问题？
3. 在 Visual Prolog 中，数据表示有何特点？
4. 在进行 Visual Prolog 与其他高级语言混合编程时，在内存使用问题上需要注意哪些方面？
5. 在 Visual Prolog 中，如何调用 Win32 API ？

附录 术 语 表

本附录是术语表 (glossary)，提供 Visual Prolog 的关键术语，这些术语以其英文字母顺序给出。

A

alignment of memory (内存对齐)

通过对一个混合论域或一个列表论域声明添加对齐说明 (alignment specification) 的前缀，可以覆盖默认的内存对齐方式。其语法为 `DOM = align 1 | 2 | 4 DOMDECL`。这里，`DOMDECL` 是一个普通的论域声明。覆盖对齐方式的主要目的是，使复合对象与使用不同于 Visual Prolog 默认值对齐方式的外部代码保持兼容。

ambiguity of names (名字的歧义性)

名字的用途在其作用域内必须清楚。如果名字表示的是谓词，则该谓词参数的数目和类型必须清楚。与调用谓词有关的歧义性可通过使用限定名字避免。为了消除歧义性，类应提供该谓词的实现程序，该谓词来自于所使用的一个归结段 (resolve section) 的多重继承。一个归结限定符用于解决来自指定源的实现。

and (与)

逻辑与和逻辑或。

由两个或多个部分组成的目标被认为是复合目标，而复合目标的每部分叫子目标。用逗号 “,” 分隔子目标，可使用一复合目标以找出一种解，该解中子目标 A 和子目标 B 均正确 (一个逻辑与)。用分号 “;” 分隔子目标，也可找出一种解，该解中至少一个子目标 A 或子目标 B 正确 (一个逻辑或)。

anonymous variable (匿名变量)

当变量绑定的值不重要时，在一个普通变量位置使用变量 ‘_’。一个以下划线开头的变量如 “_AnyName” 如果在子句中只使用一次，同样被 Visual Prolog 编译器认为是匿名变量。

arguments (参数)

在一个谓词或谓词值调用中传递的值和变量的集体名字。

arithmetic expressions (算术表达式)

算术表达式由操作数(数字和变量)、运算符(+, -, *, /)、内部数学函数 `div` 和 `mod`、括号、PFC、用户定义的数学函数、常数和十进制数值的事实变量组成。表达式的值只有当所有的变量在计算期间被绑定时才能得到。计算按一定的顺序进行, 由算术运算符的优先级决定; 优先级高的运算符先行运算。

arithmetic operators (算术运算符)

算术运算符可用于任何算术运算, 如加(+)、减(-)、乘(*)、除(/)、整数除(`div//2`)和求模(`mod//2`, 整数除法的余)。当表达式中有多种算术运算符时, 乘、除和求模首先运算, 其次是加、减。当一个表达式中所有的运算符的优先级相同时, 按从左至右的顺序进行。括号内的表达式在所有其他运算符之前优先计算。

arity of predicates (谓词的变元)

一个有 N 个参数的谓词称为 N -元谓词, 或者说该谓词有 N 个变元。不同变元的谓词即使它们名字相同, 也永远是不同的谓词。以下符号要用到:

(1) `Name/N` 表示一个 N 元的普通谓词(即不是函数) `Name`。

(2) `Name//N` 表示一个 N 元的函数 `Name`。

(3) `Name/N...` 表示一个普通谓词 `Name`, 其 N 个参数后跟着省略参数(即可变数目的参数)。

(4) `Name//N...` 表示一个函数 `Name`, 有 N 个参数, 后跟着省略参数(即可变数目的参数)。

atoms (原子)

原子是符号项或者字符串项。

B

backtracking (回溯)

Visual Prolog 内部有一个求解搜索机制。当一给定的子目标计算未能成功完成时, Visual

Prolog 返回上一子目标并试着用不同的方法来满足。这就是 Visual Prolog 的回退且重试方法，称为回溯，以找到给定问题的解。

backtracking points (回溯点)

Visual Prolog 使用回退且重试的方法，称为回溯方法，以对一给定问题找出解。当 Visual Prolog 开始寻找一问题（或目标）的解时，它要在两种可能的方法之间进行选择。它在一个分支点设置一个标志（称为回溯点）并选择第 1 个子目标进行追踪。如果该子目标失败，Visual Prolog 将回退到回溯点并尝试备选的子目标。

binary domain (二进制论域)

Visual Prolog 有一种特殊的二进制内部论域以保存二进制数据，还有用于创建二进制项和访问二进制项单个元素的谓词，二进制项的主要作用是保存数据，这些数据再没有其他合理的表示方法，如屏幕位图数据。二进制项可以以文本格式读写。二进制项可像其他项一样比较、合一。

bound variable (绑定变量)

绑定或实例化的变量，是一个引用了已知值（项）的变量。谓词 `bound/1` 可用于检查一个指定变量是否绑定到某一值。参见合一（unification）。

built-in domains, predicates, and constants (内部论域，谓词和常量)

Visual Prolog 包含嵌入的隐含类，它对内部常量、论域和谓词提供声明和实现程序。这些内部常量、论域和谓词既可用在编译中（例如，在 `#if ...` 结构中）也可用于运行中。每一编译单元隐含着这一嵌入隐含类的声明，但实际上，这一类有着在代码中不能明确引用的特殊的内部惟一名字。可在内部项的名字前使用 `::`。

C

calling a sub-goal (调用子目标)

表示 Visual Prolog 正在试着满足某子目标（属于给定谓词）的表达式。

calling conventions (调用约定)

调用约定确定了参数等如何传递给谓词。它也确定了如何从谓词名得到链接名。作为

prolog 默认调用约定的选择，可使用语言关键词 `language` 来规定调用约定为 `c`，`stdcall` 或 `apicall`。

char（字符）

`char` 内部论域。该论域的值为 `unicode` 字符，由两个无符号字节组成。在语句构成上，它可写作用单引号引用的字符 ‘a’ 或一转义顺序字符如 ‘\t’。

child class（子类）

从父类或超类继承而来的类。子类 (`child class`) 和支类 (`sub class`) 对父类而言是一样的，也称子类由父类继承而来。一个子类只能通过它的公共接口访问它的父类，即在使用父类上，它也没有比其他类更多的特权。

class（类）

每个类都有一个声明部分和实现部分。一个类可以创建与接口相应的对象，该接口在类声明的结构类型中指定。任何对象都是由类创建的。如果一个对象是由一个类创建的，而该类使用接口 `C` 来建立对象，则称为“`C` 对象”。从编程的观点来看，类是主要项：代码包含在类中。接口仅存在于程序的文本表示中；不存在（直接的）接口的运行时间表示。

class members and state（类成员和声明）

由某一类建立的所有对象都具有相同的对象成员谓词集合，但每一对象有自己的声明。因而，对象成员谓词实际是类的一部分，而对象声明是对象本身的一部分。一个类也可以包含用关键字 `class` 声明的其他命名谓词和封装声明，分别称为类成员和类声明。类成员和类声明存在于每个基类之上。对象成员和对象声明存在于每个对象基础之上。类声明可以通过类成员和对象成员访问。

clause（子句）

某一特定谓词的事实或规则，后面跟一句点 ‘.’。

comments（注释）

一句注释是一系列字符，编译器视其为单个空格字符，或者就被忽视。注释用来为代码提供资料。注释可出现于任何空格可出现的地方。既然编译器视其为单个空格，因此不能将注释放在一个记号内。在 `Visual Prolog` 中有两种注释可供使用：多行注释和单行注释。

多行注释以符号 `/*`（斜杠，星号）开头，其后是任意顺序的字符（包括新行），最后以

*/ (星号, 斜杠) 结尾。这些注释可以有多行。可以嵌套。

单行注释以符号% (百分号) 开头, 后面可跟任意顺序的字符。单行注释到行尾结束。

comparison (比较)

两项可用关系操作符比较: $>$, $<$, $>=$, $<=$, $<>$, $><$ 和 $=$ 。关系操作符是公式, 以表达式作为参数。首先, 计算左边的项, 然后计算右边的项, 最后比较结果。

compilation unit (编译单元)

编译单元是一段代码, 可由编译器单独编译生成目标文件。这样的目标文件链接到一起生成程序目标文件。一个程序只能包含一个目标段, 这是程序的入口。一个编译单元是一系列编译项。一个编译项可以是一个接口、一个类声明、一个类实现、一个目标段或一个条件编译项。

compiler directive (编译器指令)

编译器预处理程序指令。这些指令不是 Visual Prolog 语言的一部分。每个编译器指令以 # 字符开头。有 3 种编译器指令, 条件编译指令: `#if`, `#then`, `#else`, `#elseif`, `#endif`; 源文件包含指令: `#include`; 编译时间信息指令: `#message`, `#error`, `#requires`, `#orrequires`。

compound goal (复合目标)

一个至少包含两个子目标的目标。一个目标由两个或更多的部分组合而成为复合目标, 复合目标的每一部分称为子目标。可以通过用逗号 “,” 将子目标隔开, 在子目标 A 和子目标 B 都为 “真” (逻辑与) 的情况下, 找到复合目标的解。也可以通过用分号 “;” 将子目标隔开, 在子目标 A 或子目标 B 为 “真” (逻辑或) 的情况下, 找到复合目标的解。

compound item (复合项)

一个由算符和子项列表 (表中子项用逗号隔开, 并用圆括号括起) 组成的项。

conjunction (连接词)

指逻辑与 (and) 和逻辑或 (or)。

一个目标由两个或更多部分组成称为复合目标, 复合目标的每一部分称为子目标。可以通过用逗号 “,” 将子目标隔开, 在子目标 A 和子目标 B 都为 “真” (逻辑与) 的情况下, 找到复合目标的解。也可以通过用分号 “;” 将子目标隔开, 在子目标 A 或子目标 B 为 “真” (逻辑或) 的情况下, 找到复合目标的解。

constants（常量）

有名的符号常量可在常量段定义，可在现有作用域内使用。每一常量定义包括定义一个命名的常量、它的论域和它的值。若论域是标准论域之一，可省略。符号常量可用于任何地方，在这些地方也可以使用同一论域的文字。

constructors（构造器）

构造器用于构造对象。构造器可以缺省，也可以在类声明和类实现的构造器段明确声明。若一个类不声明任何构造器，那么一个默认构造器 `new//0` 被隐含声明。每一构造器有两种用途：

- （1）一个类函数，返回一个新构造的对象。
- （2）一个对象谓词，在初始化继承对象时用到。

conversions of object types（对象类型转换）

因为一个对象有其支持的任何接口的对象类型，因此它可以被用作任何这些类型的对象。也就是说，一个对象的类型可以转换为任何其支持的对象类型。在很多情况下，这样的转换是自动进行的。所以在不同的上下文中，同样的对象可以被看成具有不同的类型。可以看到对象的类型叫做可视类型，构造该对象的类的类型叫做构造类型或定义类型（参见类型转换，显式转换和隐式转换）。

cut（截断）

截断（cut）在程序中用感叹号“!”来表示。

截断迫使 Visual Prolog 在评估包含该截断的谓词时提交截至目前所做出的所有选择。一旦截断作为一子目标被评估，Visual Prolog 就不能越过它进行回溯。

D

database（数据库）

Visual Prolog 内部事实数据库是由事实组成的，这些事实可以在运行时从程序中直接加入或除去。可在事实段声明描述内部数据库事实和事实变量的谓词，像调用普通谓词一样调用这些数据库事实。但和普通谓词不同的是，可以使用谓词 `assert` 和 `retract` 在运行时加入数据库事实和除去已有的事实。可命名一个事实数据库，这同时就隐含定义了一个额外的复合论域。该论域与事实段名字相同。

database predicates（数据库谓词）

数据库谓词能在程序执行时从 Visual Prolog 系统中添加或删除事实。

declarations（声明）

一个声明引入一个名字并陈述该名字的一些性质。例如，“ X 是一个整型数”就是一个声明；它引入名字 X 并说明 X 是整型数的性质。声明的目的是引入一些名字和该名字的性质，以便能够从上下文关系中引用（即使用）该名字，并不需要知道该名字的定义。

default constructor（默认构造器）

若一个类不声明任何构造器，那么一个默认构造器已经隐含地声明了。一个默认构造器是名为 `new` 的空变元构造器。

definitions（定义）

一个定义也引入一个名字并说明该名字的确切意义。例如，“ X 等于 7” 就是一个定义，它引入名字 X 并说明该名字的确切意义，也就是 7。声明的目的是为了定义一个名字的确切意义，所以它确实有意义（在运行时）。但既然一个定义也是一个声明，所以声明的用途也可用于定义。

determinism（确定性）

大多数语言都是确定性的。也就是说，任何一套输入值将导致一套用于产生输出值的指令，并且被调用的函数仅能产生一套输出值。另一方面，Visual Prolog 自然也支持基于非确定性谓词的非确定性推理。Visual Prolog 有一套强制类型的确定性系统。确定性检查推理主要处理程序优化。Visual Prolog 确定性检查系统强制声明谓词的下列行为：谓词调用是否可以失败和谓词可以产生的解的数目。在更多 Prolog 程序执行期间，确定性模式定义：

- （1）谓词可以失败吗？
- （2）谓词可以成功吗？
- （3）对谓词调用是否要设置回溯点。

disjunction（逻辑和）

逻辑与（`and`）和逻辑或（`or`）。

一个目标由两个或更多部分组成成为复合目标，复合目标的每一部分称为子目标。当

子目标 A 与子目标 B 都为“真”，可使用一复合目标来寻找解，只要用逗号“,”将子目标隔开。当子目标 A 或子目标 B 至少有一个为“真”时，也可使用一复合目标来寻找解，只要用分号“;”将子目标隔开。

domain（论域）

在传统 Prolog 中只有一种类型——项。Visual Prolog 也有项，但必须声明谓词参数的论域是什么。论域能够使看起来相像然而种类不同的数据有不同的名字。在 Visual Prolog 程序中，关系中的项（即谓词的参数）属于论域。这些论域可以是预先确定的论域或用户特殊声明的论域。论域声明有两个非常有用的目的。首先，可以给论域冠以有意义的名字，即使它们与内部已存在的论域相同。其二，特殊的论域声明可用来声明标准论域没有定义的数据结构。

E

ellipsis（省略号）

省略号“...”可在谓词和谓词论域声明中作为最后一项形式参数。在这种情况下，它意味着声明的谓词（谓词论域）可以有一可变数目的参数。省略号流必须与一个省略参数匹配，因此只能是流模式中的最后一个流。

encapsulation（封装）

封装是一个对象隐藏其内部数据的能力和仅使其可以访问的部分能经过编程访问的方法。封装和模块的重要性广为人知。封装因像黑盒一样处理对象而使程序结构化更好、可读性更强。考虑一复杂问题，找到一个可以声明和描述的部分。将其封装进一个对象，建立一个接口，一直这样下去，直到可以声明所有的子问题。当一个问题的对象得到封装，并确保能正确运行时，就可以从它们之中进行提炼。

evaluation（评估）

一个 Prolog 程序的评估就是寻找“解”。寻找解的每一步都可能成功或失败。

exception handling（异常处理）

异常处理系统的基础部分基于两个内置谓词 `errorExit/1`（产生一个例外）和 `trap/3`（对某一计算设置异常处理程序）。当 `errorExit/1` 被调用时，当前活动的异常处理程序也将得到调用。该异常处理程序在其原始文本中执行，也就是说在它被设置的地方而不是异常发生

的地方执行。

expressions (表达式)

一个表达式是一系列表明一次计算的操作符和操作数。通常一个表达式在运行时间计算得到一个值（参见算术表达式）。

external resolution (外部分解)

一个谓词外部分解说明该谓词并不是完全在类内部实现，而是在一个外部库中实现。外部分解只能用于类谓词，即对象谓词不能被外部分解。

F

fact variables (事实变量)

一个事实变量与带有一个参数的单事实相似。然而在语法构成上，它用作可变变量（即赋值）。一个赋给事实变量的值应是一个能在编译时计算得来的项。

facts (事实)

事实是对象之间的联系。在一个事实 `likes ("John", "Mary")` 中，`likes` 是关系的名字，`John` 和 `Mary` 是元素（参见数据库谓词）。

facts database (事实数据库)

Visual Prolog 内部事实数据库由能在运行时从程序中直接加入和除去的事实构成。可以在事实段声明描述内部事实数据库的谓词，并可以像普通谓词一样调用这些数据库。但和普通谓词不同的是，可以使用谓词 `assert` 和 `retract` 在运行时加入数据库事实和除去已有的事实。可命名一个事实数据库，这就同时隐含定义了一个额外的复合论域。该论域与事实段名字相同。

fail (失败)

Visual Prolog 不能满足的一个子目标。`fail/0` 和 `succeed/0` 是两个内部的空变元谓词(built in nullary predicates)。`fail/0` 总是失败，而 `succeed/0` 总是成功，除此之外这两个谓词没有任何作用。谓词 `fail/0` 使一个谓词失败，因而总是引起回溯。

finalization (结束)

一旦一个对象不能被程序达到, 那么它可被结束 (从内存中除去), 语言的语义没有确切说明该对象何时将被结束。可以保证的是只要它能从程序中达到, 它就不会被结束。

flow pattern (流模式)

根据谓词调用中的参数是用作输入 ‘i’ (即已知) 或用作输出 ‘o’ (即未知) 形成的模式。

flow variant (流变体)

若一个谓词与几个不同的流模式相联系, 相应于该谓词的程序的一个单独的内部实现程序将为每个流模式而存在。这些不同的实现叫做该谓词的流变体。

formulas (公式)

公式代表逻辑声明, 如 “数字 7 大于数字 3”。

free variable (自由变量)

尚未引用任何值的变量。若一个变量的值是自由的, 它可通过合一绑定到论域的任何值。一旦一个变量被绑定, 可通过回溯到绑定以前的点, 重新使它自由。可用谓词 `free/1` 测试该变量是自由的还是被绑定的 (参见绑定部分)。

function (函数)

返回一个值的谓词称为函数。有时, 为了强调不是函数, 称不返回值的谓词为普通谓词。

functor (算符)

一个复合论域选项的名字。

G**global (全局的)**

一个用来允许一个以上的程序模块访问的那些常量、论域和谓词的限定词。Visual

Prolog 中惟一的全局实体包括类、接口、内部论域、谓词和常量。全局名在任何作用域内都可以访问。然而可能存在全局名被局部名遮蔽的情况。全局实体可通过加上一双冒号:: 获得（没有前置的类/接口名）。

goal（目标）

目标段是程序的入口。目标段由一个子句体构成。当程序开始时，它执行程序目标。它是在程序执行中 **Visual Prolog** 试图满足的子目标的集合。当程序目标得到执行时，程序退出。

goal tree（目标树）

一种可选择的图形表示，可在对程序目标的子目标进行评估时做出。

H

head of a list（表头）

表的第 1 个元素。

heap（堆）

堆保持比较永久或不太永久的对象。它用来存储插入内部数据库的事实、符号表、文件缓冲区、图形对象等。这些区域在事实撤销、窗口关闭等事件发生后自动释放。

I

identifiers（标识符）

“标识符”是提供给程序中变量、论域、谓词、常量、事实和事实变量的名字。标识符名字在拼写和使用时必须和任何关键字区分开来。不能使用关键字作为标识符；关键字保留用作特殊用途。标识符可通过在一个变量、论域、谓词、常量、事实或事实库的声明中指明进行创建。一旦声明，可在后面的程序代码中引用相关的值。

identity of objects（对象的同一性）

每个对象都是惟一的：对象有可变的狀態，并且因为该对象的状态可借助于它们的成

谓词来观测，所以一个对象只与自己是同一的。也就是说，即使两个对象的状态是同样的，对象也不相同，这是因为可以改变一个对象的状态而不改变另一个对象。没有直接的途径访问一个对象的状态，总是靠引用一个对象来访问对象的状态。

immutable elements（不可变元素）

Visual Prolog 的类型分为对象类型和值类型。可用术语值论域来指定不可改变的元素

的论域。这里，可以说属于相应接口名论域的对象具有可变的

状态，而任何其他论域的项都是不可变的。所以，实际上值类型不是对象类型。Prolog 中的变量是不可变的；一旦它们被绑定到一个值，它们将保留该值，除非在恢复程序以前状态的过程中，经由回溯而释放该变量。否则，一个绑定的变量是不可变的。若变量包含一个对象，那么该对象仍有可变的

状态。然而，对变量绑定的对象而言，它是不可变的。

implementations（实现）

一个类实现用于提供在类声明中声明的谓词和构造器的定义，也提供被它的构造对象所支持的任何谓词定义。

infix notation（中缀符）

在要执行运算的两个值或表达式之间具有运算符的算术表达式。

inheritance（继承）

Visual Prolog 代码继承仅发生在类的实现中。Visual Prolog 有多重继承。可以通过在一个实现的特定继承段提到一个类而实现该类的继承。从中继承的类称为父类或超类。子类（child class）和支类（sub class）对父类而言是一样的，也称子类由父类继承而来。一个子类只能通过它的公共接口访问它的父类。

integral domains（整数论域）

整数论域用来表示自然数。它们分为有符号数和无符号数两类。预定义的论域整数和无符号数以处理器结构的自然长度表示有符号整数和无符号整数（即 32 位机上为 32 位）。整数论域也可有不同的表示长度。

interface name domains（接口名论域）

每个接口的声明都声明了一个与接口名对应的论域。该论域可在谓词和其他论域的声明中当作任意其他论域使用。

internal fact database（内部事实数据库）

Visual Prolog 内部事实数据库是由事实组成的，这些事实可以在运行时从程序中直接加入或除去。可在事实段声明描述内部数据库事实和事实变量的谓词，像调用普通谓词一样调用这些数据库事实。但和普通谓词不同的是，可以使用谓词 **assert** 和 **retract** 在运行时加入数据库事实和除去已有的事实。可以命名一个事实数据库，这时就隐含定义了一个额外的复合论域。该论域与事实段的名称相同。

internal goal（内部目标）

在程序的目标段硬性编码的目标。这样的目标称为内部目标。因为它们是程序源文本的一部分且以程序代码进行编译。作为对应部分，一些 Prolog 环境支持所谓的外部目标。当这些 Prolog 环境运行不含内部目标的程序时，环境会显示特殊的对话框，在运行期间可以在这个对话框中输入一个外部目标。

K

keywords（关键字）

关键字是保留字。不能将其在程序中用作用户定义名。Visual Prolog 的关键字有 **align**, **and**, **anyflow**, **as**, **bitsize**, **class**, **clauses**, **constants**, **constructors**, **determ**, **digits**, **div**, **domains**, **delegate**, **end**, **erroneous**, **externally**, **facts**, **failure**, **from**, **implement**, **interface**, **inherits**, **goal**, **guards**, **language**, **mod**, **monitor**, **multi**, **nondeterm**, **open**, **or**, **predicates**, **procedure**, **reference**, **resolve**, **single**, **supports**, **to**。

L

language（语言）

关键字 **language** 用来告诉编译器使用哪个调用协议，而且它只在向其他语言所编写的例程声明论域的时候才出现。调用协议决定参数等如何传给谓词。它也决定链接名如何从谓词名中得到。如果省略调用协议，则它默认为 **prolog**。

last call optimization（尾部调用优化）

这是 Visual Prolog 系统内部采取的行动，以减少规则中尾部递归所占用的空间和时间

资源，也称为“尾部递归优化”。

link name（链接名）

一个链接谓词名是该谓词在定义编译单元之外可引用的名字。通常链接名用于从其他语言调用一个谓词或从其他语言的外部模块调用函数以声明一个谓词。只有类谓词可以有链接名。如果链接名没有声明，那么连接名就从谓词名得到，而得到名字的方法取决于调用协议（参见调用约定）。

lists（列表）

由包含在方括号 “[]” 中零个或多个元素的集合组成的项，集合的元素之间用逗号隔开（参见列表论域）。

literal（文字）

不变的程序元素叫做“文字”。文字可分为以下几类：整数、字符、浮点数、字符串、二进制值和表。

logical operators（逻辑运算符）

由两个或多个部分组成的目标被认为是复合目标，而复合目标的每一部分叫做子目标。用逗号 “,” 分隔子目标，可使用一复合目标以找出一种解，要求该解中子目标 A 和子目标 B 均正确（逻辑与）。用分号 “;” 分隔子目标，也可找出一种解，该解中至少有一个子目标正确（逻辑或）。也就是说，在 Visual Prolog 程序中，有两个逻辑操作符，即 ‘,’（与）和 ‘;’（或）在逻辑表达式中组合子目标。

M

mode of predicates & facts（谓词与事实的模式）

大多数语言都是确定性的。相反，Visual Prolog 自然地支持基于不确定性谓词的不确定性推理。确定性监控主要处理程序优化。Visual Prolog 确定性检查系统强制声明谓词的下列行为：对于谓词或事实调用，

- (1) 谓词可以失败吗？
- (2) 谓词可以成功吗？
- (3) 谓词调用是否设置回溯点。

这些方面的哪组应用于事实的谓词由谓词或事实的模式决定。当声明一个谓词时，模

式可以省略。在执行程序内部（即对于局部谓词），所需的流（flows）和模式源自谓词的使用。在一个接口或一个类声明（即对于公共谓词）内部省略谓词模式意味着它是一个过程。

module（模块）

一个 Visual Prolog 编译单元，它具有全局声明，从而构成项目的一部分（参见编译单元）。

multiple predicate declarations（多重谓词声明）

任何谓词可以有几个声明（对于相同的变元），每个都对参数有不同的论域声明。

mutable elements（可变元素）

Visual Prolog 的类型分为对象类型和值类型。可用术语值论域来指定不可改变的元素。这里，可以说属于相应接口名论域的对象具有可变的状况，而任何其他论域的项都是不可变的。所以，实际上值类型不是对象类型。Prolog 中的变量是不可变的；一旦它们被绑定到一个值，它们将保留该值，除非在恢复程序以前状态的过程中，经由回溯而释放该变量。否则，一个绑定的变量是不可变的。若变量包含一个对象，那么该对象仍有可变的状况。然而，对变量绑定的对象而言，它是不可变的。

N

name restrictions（名字限制）

以下是加给名字的重要限制：

- （1）子句段中的符号常量名必须以小写字母开头。
- （2）变量名字必须以大写字母或下划线字符 ‘_’ 开头。
- （3）符号文件名必须以小写字母开头。

Visual Prolog 编译器不区分大小写，除非是第 1 个字符（参见名字，小写标识符名字）。

names（名字）

名字用来表示接口、类、符号常量、论域、论域算符、谓词、事实段、事实、事实变量及变量。一个名字是一连续的字母、数字和下划线字符的序列。一名字以一个字母或下划线开头，后面接零个或多个字母、数字和下划线符的任意组合，可长达 250 个字符。不能在名字中使用空格、减号、星号或斜线。Visual Prolog 的关键字是保留字，不能用作用

户定义的名字。Visual Prolog 编译器不区分大小写，除非是第 1 个字符（参见名字限制，小写标识符名字）。

names belonging to lower-case identifiers（小写标识符名字）

接口、类、符号常量、论域、论域算符、谓词、事实段、事实、事实变量和变量的名字必须是小写标识符。小写标识符以一个小写字母开头，跟着一系列字母、数字和下划线。不能在这些名字中使用空格、减号、星号或斜线。名字可长达 250 个字符（参见名字，名字限制）。

nondeterm（不确定性模式）

默认状态下，事实的确定性模式为 **nondeterm**（不确定性）。按照它们真正的性质，在事实段中的谓词通常是不确定的。因为事实可在运行期间的任意时刻加入，所以编译器通常必须假设通过回溯找到备选解是可能的。如果在事实段有一个谓词，其事实不会多于一个，那么可以在事实声明中用关键字 **determ** 开头(或者在该谓词有且仅有一个事实的情况下用关键字 **single**)。

O

object（对象）

一个对象是一套命名的对象成员谓词和一套支持的接口。对象通常也有状态，但这种状态只能通过成员谓词而观测和改变。可以说状态封装在对象内。

object members and state（对象成员和状态）

所有对象都是由某种具有相同对象成员谓词的类建立的，但每一对象都有自己的状态。因而，对象成员谓词实际上是类的一部分，而对象状态是对象本身的一部分。一个类也包含另一组命名的谓词和一个封装的状态，用关键词 **class** 对其进行声明，分别称为类成员和类状态。类成员和类状态存在于每类基础上，而对象成员和对象状态存在于每个对象基础上。类成员和对象成员均可访问类的状态。

object state（对象状态）

一个对象的状态作为事实存储在对象中。这些事实在类实现的事实段中进行声明。对每个对象而言事实是局部的（与其他对象实体一样），而在类的所有对象中，类事实（**class facts**）是共享的。

open qualification (开放限定符)**opened scopes (开放作用域)**

开放限定符可使引用类级的实体更加方便。开放段把一作用域的名字带入另一作用域,这样无需限定符即可进行引用。开放限定符对对象成员的名字不起作用,因为它们在任何情况下只能借助于一个对象进行访问。开放段只在它们所在的作用域内起作用。举例来说,在类声明中的开放段对类的实现就不起作用。

operator priority (运算符优先级)

在算术表达式中决定运算符运算顺序的层次。

operators (运算符)

在 Visual Prolog 中,可使用算术运算符、关系运算符和逻辑运算符。算术运算符是+, -, *, /, div, mod。关系运算符是>, <, >=, <=, <>, ><和=。逻辑运算符是公式,它以公式作为参数。它们都是左结合的。运算符‘;’和‘and’是同义的,运算符‘;’和‘or’也是同义的。

or (或)

逻辑与(conjunction)和逻辑或(disjunction)。

由两个或多个部分组成的目标被认为是复合目标,而复合目标的每一部分叫做子目标。用逗号“,”分隔子目标,可使用复合目标以找出一种解,其中子目标 A 和子目标 B 均正确(逻辑与)。用分号分隔子目标,也可找出一种解,其中至少有一个子目标正确(逻辑或)。

origin interface of a predicate (谓词原始接口)

一个谓词的原始接口是文字声明谓词的接口,与通过一个支持限定符间接声明的接口相反(参见接口的支持限定符)。

P**parameters (参数)**

关系中项和变量名的集合。

parent class（父类）

从中继承的类叫做父类或超类。子类和支类（sub class）对父类而言是一样的，也称子类由父类继承而来。一个子类只能通过它的公共接口访问它的父类，即在使用父类上它并没有比其他类更多的特权。

predicate domains（谓词论域）

在 Visual Prolog 中，可声明和使用谓词论域。谓词论域的值是有着相同“签名”的谓词，即相同的参数和返回类型，相同的流模式和相同（或更强）的谓词模式。

predicate values（谓词值）

谓词值是在以下意义上可视为值的谓词：

- （1）它们可作为参数传递并可从谓词和函数返回。
- （2）可存储在事实中。
- （3）可保留在变量里。
- （4）可做相等性比较。

当然，与“无格式”谓词一样，谓词值可通过合适的参量调用。谓词值是作为谓词论域的实例声明的。

predicates（谓词）

每个 Visual Prolog 事实或规则都属于某一谓词，它指定相关的关系名和关系中相关元素的类型。一个关系的符号名叫做谓词名。与之相关的对象叫做参数。在事实 likes("Bill", "Cindy") 中，关系 likes 是谓词，而对象 Bill 和 Cindy 是参数。

program sections（程序段）

可以使用程序段来声明和定义接口、类和类实现中的程序实体。通常可使用常量、论域、谓词、构造器、事实、子句和条件段。并非所有的段都可出现在各种作用域中，详情可参考接口、类定义、类实现和条件编译的描述。

punctuation marks（标点符号）

Visual Prolog 中的标点符号对编译器而言有着语法和语义上的意义，但它们本身并不规定产生值的运算。一些标点符号，无论单独还是组合，都可以是 Visual Prolog 的运算符。标点符号有；！，．# [] () :- ::。

R

real (实数)

实数论域用于表示浮点数。内部实数论域精度由处理器体系结构决定。所有的算术、比较和赋值运算都可用于实数论域的值。允许的数值范围是 1×10^{-307} 至 $1 \times 10^{+308}$ ，即 (1e-307 至 1e+308)。必要时整数论域中的值自动转换为实数论域。

reference items and domains (引用项和论域)

如果一个未被绑定的变量被从一个子目标传至另一个子目标，包含变量最终可能绑定值的论域必须声明为引用论域。这样一个论域的元素是引用项。

relation (关系)

一个名字描述的方式，以这种方式将对象集合（或多个对象和引用对象的变量）联系在一起。关系的符号名字作谓词名。与之相关的对象叫做参量；在事实 likes("Bill", "Cindy") 中，关系 likes 是谓词，而对象 Bill 和 Cindy 是参数。

repeat ... fail combination (重复...失败组合)

一种可以通过使用 Visual Prolog 回溯机制以避免尾部递归的技术。

resolve qualification (归结限定)

名字的使用在其作用域内必须清楚。若一名字表示谓词，则该谓词参数的数目和类型必须清楚。与调用谓词有关的歧义性可通过使用限定名避免。为了消除类的歧义，类应提供谓词的实现，该谓词来自所使用的一个归结段的多重继承。一个归结限定用于归结来自指定源程序的实现。

return values (返回值)

有返回值的谓词叫做函数。为了定义谓词返回一个值，应该在谓词声明时的参数列表括号后加上“->”标记和返回值的论域。在子句定义中，返回值名字（用=号为前缀）应在子句头之后和“:-”标记前直接指明。

root and universal types (根类型和通用类型)

Visual Prolog 使用一些内部类型，称为根类型和通用类型。有通用类型就意味着有包

含着值的任何类型。举例来说，一个数字文字，如 1 并没有任何特定类型，它可用作含有 1 的任何类型的值，包括实数。算术运算对其操作数要求非常宽松。可以说算术操作数以根类型作为参数。整数根类型是任何整数类型的超类型。因此，任何整数类型可转换为整数根类型，而且，既然算术操作为根类型存在，这意味着它们中任何一种都能作用于整数论域。

rule（规则）

在事实和子目标列表之间的关系，它必须满足事实为真。

S

satisfying a sub goal（满意子目标）

Visual Prolog 为任何未绑定变量选值（如果可以）的过程。在这一过程中，根据相应谓词给定的子句，子目标为真。

scope of constructors（构造器的作用域）

构造器属于这样一个作用域，在该作用域中可产生构造器段（见类声明和类实现）。

scope shadowing（作用域遮蔽）

在当前作用域内的声明遮蔽来自其他作用域的名字(除非使用显式限定符)。

scoping & visibility（作用域与可见性）

Visual Prolog 的名字只能在程序的某些作用域内使用。这一区域叫做名字的作用域。一个接口定义、一个类声明和一个类实现都是作用域。当类的构造器被调用和当该作用域的局部变量被初始化时，作用域决定了名字的可见性。作用域也决定了一个不表示全局作用域元素名字的“寿命”（参见构造器和终结。）共有如下 4 种作用域：

（1）局部 Local（私有）作用域。一个在类实现中声明的名字只能在该类的实现中访问。这些名字有在实现中声明的论域、事实、谓词、常量和子句变量。

（2）类 class 作用域。类成员名有类作用域，类成员可通过类名限定进行访问，例如 `className::ClassScopeName`。

（3）对象 Object 作用域。对象成员名有对象作用域。它们仅能在对象存在时被访问。对象成员可仅通过引用（在类实现内它被隐含）一个经限定的对象标识符进行访问，例如，`objectID:ObjectScopeName`。

(4) 全局 `global` 作用域。接口名和类名有全局作用域。全局作用域限定符`::`可用在这些名字前。预定义的论域（字符、字符串、符号、整数、无符号数、实数、二进制数、指针、布尔数、事实数据库）、常数和谓词也可以是全局的，它们也可以用“`::`”进行限制。

sections（段）

可以使用程序段来声明和定义接口、类和类实现中的程序实体。通常可使用常量、论域、谓词、构造器、事实、子句和条件段。并非所有的段都可出现在各种作用域中，详情可参考接口、类声明、类实现和条件编译的描述。

single alternate compound domains（单选复合论域）

若一复合论域由可选算符组成，则在底部表示层它被视为结构（不同于表示可选数字的第 1 个元素的联合）并有表达式，该表达式是与 C 语言中的适当结构二进制兼容的。

stack（堆栈）

Visual Prolog 将参数传递至被调用的谓词使用的内存部分。

stand-alone programs（独立程序）

能脱离 Prolog 系统运行于某一操作系统的程序。Visual Prolog 可以创建独立运行的程序。

standard（or built in）domains（标准论域或内部论域）

Visual Prolog 包含嵌入式隐藏类，它对内部常量、论域和谓词提供声明和实现。每个编译单元隐含着这一嵌入式隐藏类的声明，但实际上，这样一个类有着特殊的内部惟一的名称。可在内部项的名称前使用“`::`”。内部论域有字符、字符串、符号、整型、无符号整型、实型、二进制、指针、布尔、事实库。

state of an object（对象声明）

一个对象的声明作为事实存在对象中。这些事实在类实现的事实段中声明。对每个对象而言，事实是局部的（像其他对象实体一样），而在所有类的对象中类事实是共享的。

stronger predicate mode（强谓词模式）

谓词模式可用如下的集描述：

- (1) erroneous = ;
- (2) failure = Fail;
- (3) procedure = Succeed;
- (4) determ = Fail, Succeed;
- (5) multi = Succeed, BacktrakPoint;
- (6) nondeterm = Fail, Succeed, BacktrackPoint.

若 Fail 在集中, 表示该谓词可以失败。若 Succeed 在集中, 表示该谓词可以成功。若 BacktrackPoint 在集中, 表示谓词可返回一个活动的回溯点。如果一个集, 比方说 failure, 是另外一个集的子集, 如 nondeterm, 则说这个模式比另一个强, 即 failure 比 nondeterm 强。

sub class (子类)

从称为父类或超类的类继承而来的类。子类和支类对父类而言是一样的, 也称子类由父类继承而来。一个子类只能通过它的公共接口访问它的父类, 即在使用父类上, 它并没有比其他类更多的特权。

sub-component (子成分)

复合元素中的子元素之一。一个复合元素是一由其他子元素(称作子成分)和描述名(算符)集合组成的单一元素。子成分用圆括号括起来, 用逗号隔开。算符写在左括号前。例如, author ("Emily", "Bronte", 1818) 复合项由算符 author 和 3 个子成分组成。

sub-element (子元素)

复合元素中的子元素之一(复合论域的)。

sub-goal (子目标)

一个关系(relation)可能涉及元素或变量, Visual Prolog 必须试着满足。

succeed (成功)

Visual Prolog 不能满足的一个子目标。fail/0 和 succeed/0 是两个内部空变元谓词。fail/0 总是失败而 succeed/0 总是成功, 除此之外, 这两个谓词没有任何作用。谓词 fail/0 使谓词失败, 因而总是引起回溯。

super class (超类)

从中继承的类叫做父类或超类。子类和支类(sub class)对父类而言是一样的, 也称

子类由父类继承而来。一个子类只能通过它的公共接口访问它的父类，即在使用父类上，它并没有比其他类更多的特权。

supports qualification for interfaces（接口的支持限定）

接口被构造在支持层次上，该结构是以接口对象为根的半格（semi lattice）。如果一个对象有接口表示的类型，那么它也有任何支持接口的类型。因此，支持层次也是一种类型层次。也可以说，对象支持接口。如果接口被命名为 **X**，那么说对象是一个 **X** 或 **X** 对象。支持限定只能用于接口定义和类实现。支持限定用于两个目的：

（1）指明接口 **A** 扩展为接口 **B**，因而类型 **A** 是类型 **B** 的子类型。

（2）声明（在实现中）某一类的对象比指定为构造类型的对象“私下”有更多的对象类型。

支持是一种传递关系：若接口 **A** 支持接口 **B**，而 **B** 支持 **C**，则 **A** 也支持 **C**。

T

tail of a list（表尾）

当给定表的第 1 个元素和分隔逗号被删除时，剩下的表。

tail recursion optimization（尾部递归优化）

Visual Prolog 系统内部采取的行动，以减少规则中尾部递归所占用的空间和时间资源，也称为“尾部递归优化”。

terms（项）

严格地讲，项是 Visual Prolog 的实体。标准类型论域的一个元素、一个表、一个变量或一个复合项，即一个跟着一系列项（选择性参数）的算符。项用圆括号括起，逗号隔开。

This variable（This 变量）

一个对象谓词总是调用一个对象。该对象带有对象事实并包含在对象谓词的实现之中。对象谓词可访问该隐含对象。称该对象为 **This**。在每个对象谓词的每一子句中，变量 **This** 隐含地定义并绑定到 **This** 对象。在一个对象成员谓词的一个子句中，其他对象成员谓词可以直接调用，因为对该操作而言，**This** 是隐含包含的。超类的成员也可以直接调用，只要调用哪种方法（参见作用域与可见性）是明确的。同样，对象事实（存储在 **This** 中）也可被访问。

trail（跟踪）

Visual Prolog 用来记录引用变量的绑定和未绑定情况的内存部分。

type system（类型系统）

使关系中所有对象或在关系中用作参数的所有变量被强制属于一个论域的方法，这个论域与那些用在相关谓词声明中的论域相对应。

U

unicode（统一的字符编码）

Visual Prolog 使用 unicode 编码的“宽（2 字节）字符”。unicode 是一种将所有字符视为固定的 2 字节的软件编码方法。这一方法用作 ANSI 字符编码方法的可选方法，由于 ANSI 编码方法仅用 1 个字节表示字符，因此只能表示 256 个字符。因为 unicode 可以表示 65 000 多个字符，所以它给许多字符不能被 ANSI 编码法表示的语言提供了表示方法。Unicode 不需要使用代码页，而 ANSI 使用代码页为有限的语言提供编码，unicode 是一种在双字节字符集（DBCS）基础上做出的改进方法，DBCS 方法混合使用 8 位和 16 位字符，并仍需要代码页。“宽字符”是使用多种语言的 2 字节的代码。当今计算世界使用的大部分字符，包括技术符号和特殊印刷符号，可根据 unicode 规范被表示为宽字符。使用宽字符简化了带有国际字符集的编程。

unification（合一）

Visual Prolog 为了满足一个子目标而将该子目标与事实和规则左部相匹配，或者确定需要评估原始子目标的一个或多个子目标的过程。当一个谓词被调用时，调用参数与每一子句的头部的项合一。合一是绑定变量的过程，在这种情况下两个项变得相等，进行尽可能少的绑定(即为进一步绑定留下尽可能多的自由项)。

universal and root types（通用类型和根类型）

Visual Prolog 使用一些内部类型，称为根类型和通用类型。有通用类型就意味着有包含着值的任何类型。举例来说，一个文字，如 1，并没有任何特定类型，它可用作含有 1 的任何类型的值，包括实数。算术操作对其操作数要求非常宽松。可以说算术操作数以根类型作为参数。整数根类型是任何整数类型的超类型。因此，任何整数类型可转换为整数根类型，而且，既然算术操作对任何根类型都存在，这意味着它们中任何一种都能作用于

整数论域。

V

variable binding（变量绑定）

变量的状态——自由或绑定。

variables（变量）

变量是以大写字母或下划线（`_`）开头的名字，后面有任意个字母（大写或小写）、数字或下划线符。

单个下划线字符表示匿名变量。当变量绑定的值不重要时，可使用匿名变量。一个以下划线开头的变量如果只在子句中使用一次，也被 **Visual Prolog** 编译器认为是匿名变量。

Prolog 中的变量是局部的，不是全局的。这就是说，如果两条子句各含有名为 **X** 的变量，则这两个 **X** 就是不同的变量。若在合一中它们恰好被放到一起，就可以相互绑定，但通常它们并不相互影响。

Prolog 中的变量是不变的，一旦它们被绑定到一个值，它们就保持该值，除非通过回溯使它们重新自由。

参 考 文 献

- [1] 涂序彦. 人工智能及其应用. 北京: 电子工业出版社, 1988
- [2] 何华灿. 人工智能导论. 西安: 西北工业大学出版社, 1988
- [3] 石纯一, 黄昌宁, 王家像. 人工智能原理. 北京: 清华大学出版社, 1993
- [4] 黄可鸣. 专家系统导论. 南京: 东南大学出版社, 1988
- [5] 吴泉源等. 人工智能与专家系统. 长沙: 国防科技大学出版社, 1995
- [6] 何新贵. 知识处理与专家系统. 北京: 国防工业出版社, 1990
- [7] 曹文君. 知识系统原理及其应用. 上海: 复旦大学出版社, 1995
- [8] 童顺等. 知识工程. 北京: 科学出版社, 1992
- [9] 廉师友. 知识系统的发展现状与趋势. 石油科技理论与应用新进展. 343~354, 西安: 陕西科技出版社, 1996
- [10] 李卫华等. IBM PC机编译型PROLOG语言. 武汉: 武汉大学出版社, 1987
- [11] 焦李成. 神经网络系统理论. 西安: 西安电子科技大学出版社, 1991
- [12] 施鸿宝. 神经网络及其应用. 西安: 西安交通大学出版社, 1993
- [13] 杨行峻, 郑君里. 人工神经网络. 北京: 高等教育出版社, 1992
- [14] 涂序彦. 专家系统发展的新动向——出席第二届世界专家系统大会的汇报. 中国人工智能学会第八届年会暨第八届全国人工智能学术讨论会论文集, 杭州: 浙江大学出版社, 1994
- [15] 林尧瑞. 80年代专家系统研究进展与展望. CAAI-7全国人工智能会议论文集. 中国人工智能学会编, 1992
- [16] 史忠植. 高级人工智能. 北京: 科学出版社, 1998
- [17] 蔡希尧等. 面向对象技术. 西安: 西安电子科技大学出版社, 1993
- [18] 陆汝钤等. 专家系统开发环境. 北京: 科学出版社, 1994
- [19] 姚天顺等. 自然语言理解. 北京: 清华大学出版社, 南宁: 广西科技出版社, 1995
- [20] 王元元. 计算机科学中的逻辑学. 北京: 科学出版社, 1989
- [21] [英] 雷蒙德(著), 赵沁平(译). 人工智能中的逻辑. 北京: 北京大学出版社, 1990
- [22] Patrick Henry Winston. Artificial Intelligence. second edition Addison-Wesley Publishing Company, 1984
- [23] Claudia Marcus. Arity Corporation. Prolog Programming. Application for Database Systems, Expert Systems, and Natural Language Systems, Addison-Wesley Publishing Company, Inc, 1986
- [24] Gary S. Kahn. From Application Shell to Knowledge Acquisition System Proceedings of IJCAI'87, 1987
- [25] Nancy Gardner Margolis. Development of an Expert System for Diagnosing Problems on a Paper Machine, Proceedings of IJCAI'87, 1987
- [26] Douglas B. Lenat, Edward A. Feigenbaum. On the Thresholds of Knowledge. Proceedings of IJCAI'87, 1987
- [27] 王永庆. 人工智能原理与方法. 西安: 西安交通大学出版社, 1998
- [28] 史忠植. 高级人工智能. 北京: 科学出版社, 1998

- [29] 马玉书. 人工智能及其应用. 东营: 石油大学出版社, 1998
- [30] 陈世福, 陈兆乾. 人工智能与知识工程. 南京: 南京大学出版社, 1997
- [31] 陆汝钤. 人工智能(上、下). 北京: 科学出版社, 1995, 1996
- [32] 刘叙华. 基于归结方法的自动推理. 北京: 科学出版社, 1994
- [33] 周济, 查建中, 肖人彬. 智能设计. 北京: 高等教育出版社, 1988
- [34] 陈文伟. 智能决策技术. 北京: 电子工业出版社, 1988
- [35] 田盛丰. 人工智能原理与应用. 北京: 北京理工大学出版社, 1993
- [36] 俞瑞别, 史济建. 人工智能原理与技术. 杭州: 浙江大学出版社, 1993
- [37] 林尧瑞, 马少平. 人工智能导论. 北京: 清华大学出版社, 1989
- [38] 冯博琴. 实用专家系统. 北京: 电子工业出版社, 1992
- [39] 王士同. 模糊推理理论与模糊专家系统. 上海: 上海科技文献出版社, 1995
- [40] 施鸿宝, 王秋荷. 专家系统. 西安: 西安交通大学出版社, 1990
- [41] 史忠植. 知识工程. 北京: 清华大学出版社, 1988
- [42] 焦李成. 神经网络计算. 西安: 西安电子科技大学出版社, 1996
- [43] 何明一. 神经计算. 西安: 西安电子科技大学出版社, 1992
- [44] 刘大有, 杨轶, 陈建中. Agent研究现状与发展趋势. 软件学报. 2000年3期
- [45] Nils J. Nilsson. Artificial Intelligence A New Synthesis. 北京: 机械工业出版社, 1993
- [46] 王士同等. 人工智能中的模糊自发式搜索技术. 北京: 机械工业出版社, 1993
- [47] 吴源泉, 刘江宁. 人工智能与专家系统. 长沙: 国防科技大学出版社, 1995
- [48] 俞瑞钊, 史济建. 人工智能原理与技术. 杭州: 浙江大学出版社, 1993
- [49] 杨详金, 蔡庆生. 人工智能. 重庆: 科学文献出版社重庆分社, 1988
- [50] 周详和等. 自动推理引论及其应用. 武汉: 武汉大学出版社, 1987
- [51] 赵瑞清. 专家系统原理. 北京: 气象出版社, 1987
- [52] 钟义信等. 智能理论与技术——人工智能与神经网络. 北京: 人民邮电出版社, 1992
- [53] 张景, 李人厚. 基于时序规则的实时控制专家系统结构及实现. 第一届全球华人智能控制与智能自动化大会论文集. 北京: 科学出版社, 1993
- [54] 张景. PC系列计算机故障诊断专家系统的 PROLOG实现. 微电子学与计算机. 1990年第9期
- [55] 张景, 李人厚. 混合使用专家系统与神经网络的智能控制系统. 系统工程与电子技术. 1994年第12期
- [56] 张景, 李人厚. 专家系统规则库——神经网络翻译算法. 微电子学与计算机. 1996年, 第5期
- [57] Xiaorong Huang (黄小戎), Natural language generation: theories and applications, 全国第三界计算语言学联合学术会议, 1995
- [58] Yoshiyasu Takefji, Neural Network Parallel Computing, Kluwer Academic Publishers, 1992
- [59] Yun LI et al. Genetic algorithm automated approach to the design of sliding mode control systems, Int. J. Control, 1996, Vol. 63, No. 4
- [60] Zhang jing, Ding Julan. The Implementation of Fault Diagnostic Expert System for Personal Computer, Journal of System Engineering and Electronics, Vol. 8, No. 8, 1997
- [61] Zhang jing, Li renhou, Neural Network Expert System and Their Application, Pacific-Asian Conference on Expert System, Proc. On PACES'95, 1995
- [62] 蔡自兴. 人工智能研究发展展望, 高技术通信. 5(70): 59~61
- [63] 蔡自兴, 徐光佑. 人工智能及其应用(第二版). 北京: 清华大学出版社, 1996

- [64] 傅京孙, 蔡自兴, 徐光佑. 人工智能及其应用. 北京: 清华大学出版社, 1987
- [65] 胡状麟, 朱永生, 张德录. 系统功能语法概论. 长沙: 湖南教育出版社, 1987
- [66] 胡运发. 人工智能系统——原理及设计. 长沙: 国防科技大学出版社, 1988
- [67] 候广坤等. 人工智能概论. 广州: 中山大学出版社, 1993
- [68] 胡守仁. 神经网络导论. 长沙: 国防科技大学出版社, 1996
- [69] 韩楨祥, 文福全. 模拟进化优化方法及其应用. 计算机科学. Vol. 22, No. 2, 1995
- [70] 曾黄麟. 粗集理论及其应用. 重庆: 重庆大学出版社, 1996
- [71] 金志权等. 人工智能程序设计. 南京: 南京大学出版社, 1986
- [72] 林尧瑞, 张钱, 石纯一. 专家系统原理实践. 北京: 清华大学出版社, 1988
- [73] N. J. 尼尔逊. 人工智能原理. 北京: 科学出版社, 1991
- [74] 陆汝钤. 人工智能(上、下) 北京: 科学出版社, 1989, 1996
- [75] 刘叙华. 基于归结方法的自动推理. 北京: 科学出版社, 1994
- [76] R. S. 迈克尔斯基等. 机器学习——实现人工智能的途径. 北京: 科学出版社, 1992
- [77] Peter Coad, Edward Yourdon. 面向对象的设计. 北京: 北京大学出版社, 1994
- [78] 潘卫东. 利用遗传技术设计人工神经网络. 模式识别与人工智能. 1994年1期
- [79] 渠川路. 人工智能: 专家系统及智能计算机. 北京: 北京航空航天大学出版社, 1991
- [80] 石纯一, 吴轶华编译. 人工智能基础NCIC-YM91, 1991
- [81] 王士同. 模糊推理理论与模糊专家系统. 上海: 上海科技文献出版社, 1995
- [82] 赵瑞清, 王晖, 邱逢虹. 知识表示与推理. 北京: 气象出版社, 1991
- [83] Joseph Giarrano, Gary Riley. 专家系统原理与编程(英文版, 第3版). 北京: 机械工业出版社, 2002
- [84] 王万森. 人工智能原理及其应用. 北京: 电子工业出版社, 2000
- [85] 邵军力, 张景, 魏长华. 人工智能基础. 北京: 电子工业出版社, 2000
- [86] 武波, 马玉祥. 专家系统. 北京: 北京理工大学出版社, 2001
- [87] 廉师友. 人工智能技术导论. 西安: 西安电子科技大学出版社, 2000
- [88] George F Luger. 人工智能: 复杂问题求解的结构和策略(英文版, 第4版)(Artificial Intelligence: Structure and Strategies for Complex Problem Solving, Fourth Edition.). 北京: 机械工业出版社, 2003
- [89] 陆汝钤. 知识科学与计算科学. 北京: 清华大学出版社, 2002
- [90] 雷英杰, 邢清华, 孙金萍, 张雷. Visual Prolog智能集成开发环境评述. 空军工程大学学报(自然科学版), 2002, Vol. 3, No. 5
- [91] 雷英杰, 张雷, 邢清华, 孙金萍. Visual Prolog 语言教程. 西安: 陕西科学技术出版社, 2002.
- [92] 雷英杰, 邢清华, 孙金萍, 张雷. Visual Prolog 编程、环境及接口. 北京: 国防工业出版社, 2004.
- [93] 雷英杰, 王涛, 赵晔. Visual Prolog的回溯机制分析. 空军工程大学学报(自然科学版), 2004, Vol. 5, No.5
- [94] 雷英杰, 王宝树, 赵晔, 王涛. Visual Prolog的搜索控制机制分析. 计算机科学, 2005, Vol. 32, No. 4
- [95] 雷英杰, 华继学, 徐彤, 狄博. Visual Prolog截断机制对回溯的作用机理. 计算机工程, 2005, Vol. 31. No. 18
- [96] 邢清华, 雷英杰, 刘付显. 一种按比例分配冲突度的证据推理组合规则. 控制与决策, 2004, Vol. 19. No. 12

-
- [97] 雷英杰, 王宝树. 拓展模糊集之间的若干等价变换. 系统工程与电子技术, 2004, Vol. 26. No. 10
- [98] Xing Qinghua (邢清华), Lei Yingjie (雷英杰), Liu Fuxian (刘付显). Study on knowledge processing techniques in air defense campaign intelligent aid decision. ICCIMA' 03 Proceedings by IEEE Press, 2003